MASTER IN ARTIFICIAL INTELLIGENCE

MASTER THESIS

Deep 3D Pose Regression of Real Objects Trained With Synthetic Data

---

Author:
Pau BRAMON MORA

Supervisor:
Dr. Sergio ESCALERA GUERRERO

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) - Barcelona Tech

Facultat de Matemàtiques de Barcelona
Universitat de Barcelona (UB)

Escola Tècnica Superior d'Enginyeria
Universitat Rovira i Virgili (URV)

April 24, 2019

MASTER IN ARTIFICIAL INTELLIGENCE

Abstract

Deep 3D Pose Regression of Real Objects Trained With Synthetic Data

by Pau BRAMON MORA

Deep Learning has achieved outstanding results in several applied computer vision areas. However, for manufacturing settings, it is still complicated to use Deep Learning techniques, because these applications usually lack of sufficiently large datasets. In this thesis, we generate synthetic objects under different 3D configurations and visual point of views to train deep models to detect and regress 3D poses of associated real objects in manufacturing settings. In particular, the approach is to use domain randomization in the synthetic generation to bridge the reality gap, so that the proposed deep model can generalize to real images in test. Moreover, this work proposes a new multi-task Neural Network for the pose regression task, which uses a new loss function to reduce the errors produced by view ambiguities and symmetries.

The results obtained in this work prove that it is possible to obtain good results in the pose regression task exclusively using synthetic images. Using the architecture and the generation pipeline defined in this thesis, the prediction of the location and regression of different objects can be regressed fairly accurately.

Contents

Abstract **i**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Related Work . . . 2
1.2.1 Datasets . . . 2
1.2.2 6D Pose Regression . . . 3
1.2.3 Bridging the Reality Gap . . . 7
1.3 Description of the System . . . 8

2 Datasets **10**
2.1 Synthetic Data . . . 10
2.1.1 Reality Gap . . . 11
2.1.2 Synthetic Data Generation . . . 13
2.1.3 Domain Randomization . . . 15
2.2 Dataset Creation . . . 17
2.2.1 6D-Labeler . . . 17
2.2.2 6D-Single Object Dataset . . . 20

3 Network **21**
3.1 Overview of the Network . . . 21
3.2 Multi-task network . . . 23
3.2.1 Bounding Box Prediction . . . 23
3.2.2 Rotation Prediction . . . 24
3.2.3 Location Prediction . . . 25
3.3 Projection Loss . . . 25
3.3.1 Dealing with view ambiguity and symmetries . . . 25
3.3.2 Definition of the Projection Loss . . . 27
3.3.3 CUDA Implementation . . . 36
3.4 Implementation Details . . . 40
3.4.1 Network Details . . . 40
3.4.2 Data Augmentation . . . 41

4 Results **42**
4.1 Evaluation Method . . . 42
4.2 Ablation Study . . . 43
4.2.1 Effect of the domain randomization . . . 43
4.2.2 Effect of the Projection Loss . . . 47
4.3 Results on the created dataset . . . 50

5 Conclusion **54**
5.1 Summary of Contributions . . . 54
5.2 Directions for Future Work . . . 55

Bibliography **57**

Acronyms

6DSO 6D-Single Object Dataset. 19, 42, 43, 47, 50, 51

CNN Convolutional Neural Network. 3, 4

DL Deep Learning. 1, 2, 6, 8

DNN Deep Neural Network. 1, 3, 9

GAN Generative Adversarial Network. 8, 55

GPU Graphics Process Unit. 36

ICP Iterative Closest Point. 3, 6

IoU Intersection Over Union. 27, 29, 30, 31, 46, 49

NN Neural Network. 1, 8, 40, 43

PNG Portable Network Graphics. 12, 15

PnP Perspective-n-Point. 4, 10, 17, 18, 19

ReLU Rectified Linear Unit. 23, 24, 25

RoI Region of Interest. 5, 21, 24, 43, 45, 47

RPN Region Proposal Network. 5

SSD Single-Shot Detector. 3

YOLO You Only Look Once. 4

Chapter 1

Introduction

1.1 Motivation

The industrial automation sector is rapidly evolving every day and adopting new technological trends. A promising technique that is becoming very popular in automation companies is Deep Learning (DL). DL and, particularly, Neural Networks (NNs) have proven very effective in many tasks, achieving even human-like performance in some of them.
Computer Vision is one of the areas where DL has achieved great success. Deep Neural Networks (DNNs) solve vision problems with a much higher level of abstraction than classical computer vision algorithms, achieving in some tasks astonishing results. It is not surprising that computer vision companies in the automation sector are starting to use DL as one of their tools for tasks such as quality inspection and object classification.

Regressing the 3D location and orientation of objects (also known as 3D pose or 6D pose regression, which will be used interchangeably throughout this work) is a crucial task in many manufacturing settings. Many robotic applications (such as robot manipulation or inspection tasks among others) need to know the translation and rotation of objects to be performed. However, the 6D pose regression task is extremely challenging. Objects can have very complex shapes, many ambiguous viewpoints and the light conditions can totally change the projected shades and reflections in the obtained image. Considering the success in other tasks, some see in DL a promising way to tackle this complex problem.

The main concern in the use of DL for real applications is the availability of data. It is well-known that training deep models needs huge amounts of annotated data, which sometimes are very hard to obtain. Without the proper data, it is not possible to train deep models and, in real applications, there usually are not many available datasets. This is specially problematic in the 6D pose regression task, where obtaining images with the ground truth location and rotation tends to be very complicated. This is probably the reason why computer vision companies still rely on complex hardware like stereoscopic 3D cameras or RGB-D sensors to perform this concrete task.

The goal of this project is to create an end-to-end system able to regress the 6D pose of different objects for industrial automation applications using DL techniques. In order to solve the task without available datasets, the system will synthetically generate its own training data to train a DNN. Therefore, for a given object, the system first generates a dataset of synthetic images and then it trains a Deep Model with it.

This master thesis aims to prove that the proposed system can achieve good results in pose regression tasks and that this methodology could be useful in real world industrial applications.

Probably the most challenging part of the project is to obtain good results with real images using a model trained exclusively with synthetic ones. The concept of *reality gap* refers to this challenge and is widely studied in the literature [45, 47, 41]. Even though this is a complex problem to solve, the idea of focusing on manufacturing settings makes the system feasible. Manufacturing configuration settings have the following properties that make them more suitable for this system:

- The environment in industrial applications has a lot less variability than in other applications.
- The objects to regress are very similar to the 3D models used to generate the images, making it possible for the render to generate fairly realistic synthetic datasets.
- We can moderately control the environment, meaning that we can add elements that could help to regress the pose more accurately (adding references, choosing the right light conditions, using specific background colors, etc.).

This work also aims to obtain a system computationally feasible. The goal is to design a system that could be used in real applications, therefore the computational times of the synthetic generation process have to be also practicable. Throughout this project, this was kept in mind, comparing the computational time and the complexity when choosing the right generation procedure.

The DL model proposed in this master thesis is a multi-task Neural Network. In particular, the network is composed by a main feature extractor followed by three branches regressing the bounding box of the object, the translation with respect to the camera and the rotation with respect to a reference position. Furthermore, the architecture presented in this thesis also introduces a new loss function to solve the problem of view ambiguities.

1.2 Related Work

This section analyses other papers that are somehow related to the work presented in this master thesis. First of all, we explore some of the datasets available for the 6D pose regression task and the available tools to create new synthetic datasets. Secondly, we summarize some of the most relevant papers working on the 6D pose regression task using DL techniques. Finally, we focus on the literature addressing the *reality gap* problem when using synthetic data to train deep models.

1.2.1 Datasets

Since object detection and pose estimation has become a very popular problem to solve using Deep Learning techniques, there are plenty of available datasets in the literature to test proposed solutions in different scenarios [22, 21, 29, 6]. However, the aim of this work is to create an end-to-end system able to solve completely new problems in manufacturing settings. Consequently, the idea is to create a specific

dataset for simulating an industrial setting, instead of using any of the available ones for generic problems. Furthermore, the data used to train the deep models will be synthetically generated using a rendering software.
There are many available tools to create images from 3D models (e.g. [46, 49, 28, 35]). In this work, we will use Blender [5], because it is open-source, has a python interface and it has a large range of possibilities for rendering (from very simple but fast renders to very complex but computationally expensive ones).

1.2.2 6D Pose Regression

Many recent papers tackle the problem of regressing the 3D location and rotation using deep models. In this section we describe some of the most relevant ones at the time of writing. At the end of the section, a comparison of all the models is given for the most used dataset in the literature: the LINEMOD dataset [21].

SSD-6D

The interesting work presented in [26] explores the use of a modified version of the Single-Shot Detector (SSD) network [33] to obtain the 2D detection of objects and infer fairly accurate 6D poses of them. In a second step process, they refine the 6D poses using either the RGB images with an edge-based approach or RGB-D data with a cloud-based Iterative Closest Point (ICP) approach. The main idea is treating the pose regression as a classification problem with discrete number of viewpoints and in-plane rotations. Interestingly, in this work they also train the DNN exclusively with synthetic images.

The architecture described in this work uses an InceptionV4 network [43] with pre-trained weights as the backbone of an SSD network, from which they obtain six different feature maps to apply the method at six different scales. Following a fully convolutional SSD like method, they can obtain, at each scale and location in the image, different tensors describing the object class, discrete viewpoint, discrete in-plane rotation and a refinement of the four bounding box corners. Therefore, instead of directly predicting the orientation, they treat the pose regression as a classification problem with a discrete number of viewpoints and in-plane rotations. This network is trained using synthetic images, generated using a 3D model over a random background image (using randomly selected images from the MS COCO dataset [32]).

The results from the network provide, for each object in the image, a pool of 6D hypothesis with the most confident viewpoints and in-plane rotations. In this second step, they use the 6D hypothesis with its bounding box to render the 3D models and compare the obtained projections with the actual test image. Using either the RGB image alone or also the RGB-D data, an accurate pose can be inferred.

BB8

In the work presented in [36], they propose a three-stage process to regress the pose: first they find the location of the objects in 2D, secondly they perform an estimation of the 3D poses and, finally, they refine them. For the localization step, instead of searching for the bounding boxes, they use an object segmentation approach.

Once they have the objects in the image isolated, they use another network to find the bounding box points of each object. Using a PnP algorithm with the regressed bounding box points, they can infer the pose of the object in 3D, which can then be refined using another Convolutional Neural Network (CNN) proposed in the paper. A very interesting contribution of the work is the strategy to deal with symmetric objects, which is a common problem in most of the papers solving this task. In the paper, they achieve remarkable results in the LINEMOD [21] dataset and the challenging T-LESS [22] dataset, which contains complex symmetric objects.

In the localization step, instead of using standard 2D object detection and localization methods, they use an object segmentation approach. This segmentation is done as a two-level, coarse-to-fine, object segmentation. The first level uses a network based on a VGG architecture [42] to obtain a low resolution segmentation of each object. The second level is a simple 2-layer CNN that obtains a finer segmentation for each object separately.
In the second part, they use the segmentation obtained previously to isolate the areas containing the different object in the scene. Using a modified version of a VGG network, they are able to regress the pose of each object separately. This regression is tackled as a regression of 3D-to-2D correspondences. The network first regresses the 2D coordinates of the 3D bounding boxes and, with a Perspective-n-Point (PnP) algorithm, they are able to infer the 3D pose of each object.
Finally, as a separate step, they propose another network to perform a refinement of the poses. The proposed refinement is an iterative process using a network specific for each object class. This network uses the area of the image containing the object and a mask generated from the current estimation of the pose. This network will produce at each iteration an update of the current estimation of the pose.

Real-Time Seamless Single Shot

While the above papers require a second step process to refine the obtained pose, the work presented in [44] proposes a single-shot deep CNN to directly obtain an accurate prediction without a posteriori refinement. Their implementation is based on You Only Look Once (YOLO) [39] and it is able to directly predict nine 2D coordinates (the eight bounding box corners and the centroid of the object's 3D model) from which they can infer the 6D pose with a PnP algorithm.

They follow an architecture similar to the fully convolutional YOLO v2 architecture presented in [38], that takes a single color image and divides it into a regular grid producing SxS cells. Each location of the grid will provide the 2D coordinates of the 9 points (bounding box and center), an overall confidence value and the class probabilities. For the objects with a certain confidence value, they can directly regress the pose of the objects with a simple PnP algorithm using the 9 points found in 2D.
In order to improve the results, they us passthrough layers to use features from earlier layers in the prediction and they train the network with different input resolutions (choosing multiples of 32, since the network downsamples the image by a factor of 32).

PoseCNN

The papers described above find points in the 2D image to do the pose regression using a PnP algorithm and the 3D model. Alternatively, the work presented in [50] proposes a network that is able to directly predict the location and rotation of multiple objects in the scene. In order to accomplish this, the network implicitly performs three different tasks: an object segmentation task to find all objects in the scene, a prediction of the 3D translation vector and a regression of the quaternion describing the rotation of the object.

The network consists of a base network of 13 convolutional layers and 4 max-pooling layers, followed by the three different branches that perform the three tasks described before. The first branch of the network does a semantic segmentation of the image, labelling each pixel as one of the objects in the scene. The information obtained in this branch is then used in the following ones to perform the 6D pose estimation.
The second branch aims to find the 3D translations of the different objects within the image. In this part, they first localize the centres of each object instance and estimate its depth (distance from the camera). Using this information, the network can easily infer the 3D translation and the bounding boxes of each object. This branch uses a Hough voting layer to obtain the 3D location of the different objects. Using this layer, they claim that the regression is more robust to occlusions than directly predicting a vector with the position.
Finally, the last branch uses a Region of Interest (RoI) pooling layer [17] with the bounding boxes found before to extract the features describing the object. This way, they regress the rotation using only the features contained in the are of the object, making branch smaller. Using fully connected networks, the branch directly outputs a quaternion representing the rotation of the object with respect to a known orientation.
Furthermore, in order to deal better with symmetric objects, they propose an special loss function that compares volumes. In training, they combine all the different losses of the network and they train it end to end.

Deep-6DPose

Inspired by the Mask-RCNN [20] implementation for object instance segmentation, the work presented in [11] proposes the use of Region Proposal Network (RPN)[40] in the 6D pose regression task. The network presented in this work consists of two different parts, the first one is a generic feature extractor to obtain the necessary features and the second part contains the branches performing the necessary tasks of the network. These two parts are connected using the RPN, which find the parts in the image that are more likely to contain objects.

The backbone of the network is based on the VGG architecture, which extracts the features of the image. Using RPN attached to the VGG, they obtain areas likely to contain objects. Using a RoI pooling layer, they extract the different features within the proposed areas, which are then evaluated with the second part of the network.

The second part consists of three different branches that find the necessary information for the pose regression. The first branch performs a bounding box regression and a class classification of the proposed region. This determines whether the region

actually contains an object or not, and it regresses a tighter bounding box.
The second branch finds a segmentation of the object within the proposed region. Together with the information of the bounding box and class, the objects are perfectly located within the image.
Finally, the third branch regresses a 4D vector with the necessary information to regress the final location and rotation. One element of this vector is the z component of the translation vector, which together with the bounding box and object information allow them to perfectly regress the precise translation of the object. The three remaining elements of the regressed vector are Lie algebra associated with the rotation matrix of the pose.

DeepIM

As it was already explained, in most of the literature, once a first estimation have been obtained, some algorithm to refine it can be used to improve the results. There are many different alternatives to refine the pose based on classical computer vision techniques (like edge based methods or ICP). The refinement process can be done with the same RGB images used for the prediction, but also with depth information. The work presented in [31] shows another method for refinement based on Deep Learning (DL). Here, the authors present a network to obtain a pose correction using the image, the 3D model of the predicted object and an initial estimation.

The main idea of this work is to use the original image and a synthetic render of the object in the predicted pose, to find the adjustment that makes the two images match. In order to do that, they use the two images (original and synthetic) concatenated together with two foreground masks of the two images as the input tensor of the network. Using the FlowNetSimple architecture [13] as base, the network is trained to predict the optical flow between the pair of images. This network can be used iteratively to refine the pose more precisely. Using this technique, the paper achieve large improvements on the results obtained in [50].

Comparison

Table 1.1 shows a summary of the related work presented above. In this table, the results are obtained for the LINEMOD dataset [21] and compared using the ADD metric [21]. The results shown are achieved using only RGB images. Notice that this table only shows the accuracy of the results, but not the computational time. While methods like [26, 31] obtain better results in terms of precision, others like [44, 11] claim to be faster to compute.

Apart from [26], all methods use part of dataset for training (around $15\% - 30\%$ depending on the paper). Since the dataset is not very large, strong data augmentation techniques are used in all of them (random background, rotation and scale variations, changes in illumination, etc.). Some of them also use synthetic images to enlarge the dataset.
On the contrary, the SSD-6D[26] is trained exclusively on synthetic images, obtaining remarkable results after some refinement. Notice that this network only produces 6D hypothesis without refinement, because the problem is treated as a classification problem instead of a regression. Since the output is the discretized viewpoint and

Work	Description	Refinement	Results
SSD-6D [26]	InceptionV4 for SSD approach to get bounding box hypothesis	None	2.42
SSD-6D [26]	InceptionV4 for SSD approach to get bounding box hypothesis	Edge based	76.3
BB8 [36]	Segmentation + CNN to detect bounding box corners	None	43.6
BB8 [36]	Segmentation + CNN to detect bounding box corners	CNN per object	62.7
Tekin [44]	YOLO based approach to predict bounding box corners and centroid	None	55.95
Deep-6DPose [11]	Extended Mask R-CNN for pose estimation	None	65.2
PoseCNN [50]	Semantic labelling + directly estimating location and pose	None	62.7
PoseCNN [50] + DeepIM [31]	Semantic labelling + directly estimating location and pose	DeepIM [31]	88.6

TABLE 1.1: Related work results with the LINEMOD dataset [21] using only RGB images.

in-plane rotation, it cannot find a precise output and, therefore, the results without refinement are pretty low.

1.2.3 Bridging the Reality Gap

As we will explain in detail in Chapter 2, one of the major obstacles in using synthetic data to train neural networks is the so called *reality gap*. The reality gap is a subtle but important discrepancy between real and synthetic data, that prevents neural networks trained with synthetic images from generalizing to real ones. Even though this is a relatively new and complex problem, there are some papers in the literature that have found very useful methods to close that gap and make the networks generalize.

In [45], the authors propose an alternative to improve the results in networks trained exclusively with synthetic data. It explores the use of domain randomization in simulated environments. Domain randomization consists of randomly varying the characteristics of the simulated environment, such that the network only learns the important and invariant features. With this technique, they are able to train a deep neural network to do detection and pose estimation for robotic tasks using only synthetic data. In the paper they can achieve similar results to the ones obtained with real data, but only using simulated images. In their approach, differently from others using simple backgrounds, they create coherent scenes as similar as possible to what the camera will see in real test cases. Then, they make the simulator introduce many variations so that the network learns only the important features of the scene.

Similar to the previous paper, the work presented in [47] shows how using domain randomization can achieve better results than using photorealistic synthetic images alone. In this paper, they focus only on the object detection task for the Real Image KITTI dataset [16], training different architectures with complex photorealistic images from Virtual KITTI dataset [14] or simpler ones but using domain randomization. The results show how using domain randomization actually obtains better results, achieving even better results when the network is fine tuned with a few real images.

With the same idea of closing the reality gap between models trained with synthetic images and real ones, a very interesting solution is presented in [37]. In this paper, the authors propose the use of a network to adapt features obtained with real and synthetic images. They basically add an intermediate network between a feature extractor and a pose predictor that learns how to map features of real images into the synthetic feature space. They propose to learn this mapping using pairs of synthetic and real images in similar poses.

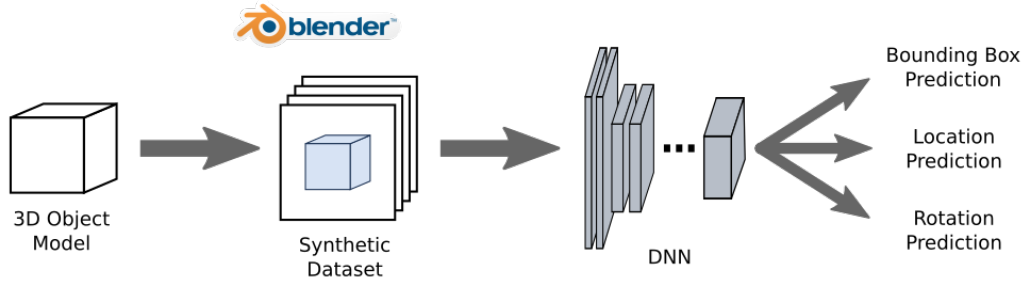


FIGURE 1.1: Overview of the system.

In the paper they show how their solution is able to improve the results in two different problems: 3D object pose estimation and 3D hand pose estimation. They use implementations from other works for the feature extractor and the pose prediction networks, but the idea of creating a network to adapt features could be used with any implementation.

One of the main problems in the solution presented in [37] is obtaining the pairs of real images and synthetic ones in similar poses. Another alternative is proposed in [41], which bridges the reality gap using only unlabeled real data. The approach here is using adversarial networks to add realism to the synthetic datasets. The advantage of this approach is that the adversarial network just needs unlabeled real data, without the costly annotations needed in the previous work.
In this paper, they use Generative Adversarial Networks (GANs) to add realism to synthetic images, in order to later use them to train other deep models. They qualitatively show how, using those models, they can generate highly realistic images. Using those improved images for training, the network is able to generalize a lot better to real images. This work demonstrates a significant improvement over using synthetic images alone, and achieves state-of-the-art results on the gaze estimation dataset MPIIGaze [51] without any labeled real data.

1.3 Description of the System

The proposed application in this work is an end-to-end system able to regress the location and rotation of objects in industrial settings using a NN. The idea is training the system only with synthetic data, making use of the domain randomization method. The synthetic datasets will be generated in Blender[5] and enlarged with simple data manipulation in Python. Furthermore a new DL architecture able to regress the 6D pose of single objects is proposed and evaluated. Figure 1.1 shows an overview of the proposed system.

The first part of the system is the generation of data. First, the work proposes the pipeline for the synthetic data generation. The pipeline should be able to produce training datasets *autonomously* and *efficiently*. Autonomously means that the system should be able to create the datasets almost automatically, without the need of human designers crafting the perfect 3D scene for each new applications. Efficiently means that the generation of images should be as fast as possible. The main reason for the use of synthetic datasets is to satisfy the rapid and constant evolution of the

industrial needs. Since factories have to constantly be able to produce new products, industrial applications are always changing and adapting. For this reason, the system should be fast and totally automatable, reducing as much as possible the rendering times and the necessary work to set up a new product or application.

Chapter 2 details the synthetic data generation pipeline and the methods used to overcome the generalization problem. In addition, this work contributes with a new annotated dataset of real images for the pose regression task and a small interface to manually create other datasets for the same problem. Even though the deep models of this work are trained with synthetic data alone, in order to test the performance of the system in real images, an annotated dataset of real images is obviously necessary. This chapter also describes the new dataset and the interface to manually create others.

The second part of the system is the design of a DNN able to correctly solve the 3D pose regression task. As it can be seen in Figure 1.1, the network predicts three different vectors: the bounding box in 2D, the location of the object with respect to the camera and the orientation of the model with respect to an initial position. The proposed network is a multitask network that will perform various operations using the same feature extractor, combining the different losses of all the tasks to obtain a more efficient training. The work also introduces a new loss function for the pose regression task, that helps to deal with view ambiguities. This loss is called Projection Loss and aims to imitate the matching of silhouettes that humans do when trying to understand the pose of an object. This new loss and the whole design of the network is studied in Chapter 3.

In Chapter 4 the generation pipeline and the proposed DNN are evaluated on the created dataset. In this chapter, different evaluations are performed. On the one hand, a qualitative evaluation is shown. The 6D regression task is an intuitive task for humans, so a visual evaluation of the results is necessary to understand the problems the system faces. As it will be observed, a visual inspection of the results provides valuable information about the effects of the different elements of the system. On the other hand, a quantitative evaluation is also performed. This evaluation will be performed using the most common metric in the literature and it will be essential to compare the different architectures or generation pipelines. Even though the qualitative results give important insights about the problems of the network, to compare different implementations some kind of metric is indispensable.

Finally, Chapter 5 gives some conclusions about the implemented system and the obtained results. In this chapter, some future work or research lines are stated, so as to give continuity to this work.

Chapter 2

Datasets

This chapter focuses on the dataset generation for both training and test. The first part is focused on the creation of training datasets. As it was stated, the work aims to train neural networks with synthetic data. Here the main generation pipeline is detailed and the main issues regarding the generalization problems are addressed. The second part is instead focused on the test datasets. In order to evaluate the performance of the network, a test dataset with real annotated images is necessary. First, the interface developed to manually annotate real images with 3D pose labels is presented. Secondly, the annotated dataset that will be used to evaluate the network in Chapter 4 is described.

2.1 Synthetic Data

Training deep networks for computer vision tasks typically requires an enormous amount of labeled training data. Consequently, obtaining good results in deep learning usually implies spending a lot of time manually annotating data. This could be fine for some applications or research experiments, but it may become a problem in many real applications.

First, some tasks are harder to label than others. In some cases, the labeling process can be very complex and time-consuming. For example, for the pose regression task, manually finding the pose of an object requires a lot of time, for example, finding the known 2D points to apply a PnP algorithm. Furthermore, training a deep model to regress the pose of an object usually requires a lot more data than a simple classification of the same object, since the task is much more complex.

Secondly, if we are implementing a system to work in the industrial automation sector, which is continuously changing and usually needs quick implementations, manually annotating data for each problem may be not the best option. The aim of this work is creating a system that could work in real applications, so we cannot rely exclusively on manually annotated data.

A promising approach to overcome this limitation is to use synthetic images instead of real data. The idea is to create datasets of artificially generated images using a simulator or a rendering software. With this approach, very large datasets could be created in a much shorter time, even for tasks with very complex labels. Consequently, we would overcome one of the main issues regarding Deep Learning, which is the availability of data.

In this work, we explore the use of synthetic data to train deep models for the pose regression task. First, this section explains the problem called *reality gap*, the main issue that arises when training neural networks with synthetic data. It exposes two ways to deal with it and the approach chosen in this work. Secondly, the section describes the generation pipeline used to create the datasets. This pipeline will be used through the rest of the work to create any synthetic dataset needed. Finally, the section focuses on the domain randomization approach and how the pipeline described before uses it. The domain randomization will be a key element in our system and its effects will be experimentally studied in Chapter 4

2.1.1 Reality Gap

Using synthetic images instead of real ones seems a promising idea, but, unfortunately, this approach does not always work as expected. If the datasets are not created carefully, the networks can end up learning just the synthetic representation and not the real one. If the images in the training dataset differ from the real ones, the network will not generalize and the results in test will be poor. The differences between the synthetic data used for training and the reality is called the *reality gap*, and it is the main problem in the use of synthetic data for deep learning. In order to bridge this reality gap, we need to either create very realistic datasets or find a way to force the network to learn just the important characteristics of the data.

If we want to create very realistic datasets, there exist some simulators and rendering tools which are able to create highly realistic scenes. Mastering these tools, we could apparently overcome the generalization problem creating datasets just like the real images. There are many examples where photorealistic images have been successfully used as training datasets for deep learning models [48, 15, 24].

However, even though photorealistic images may seem a good solution, creating large datasets of them can be extremely complex. First of all, creating very realistic images is computationally expensive. Both the hardware and the software tools for rendering are improving every day, but the time required to do it is still significant. For example, the time needed to create the image in Figure 2.1, using Blender Cycles rendering engine with a NVIDIA GeForce GTX TITAN Black 1050, is still larger than 60 seconds. This is a lot of time for a single image with such a powerful GPU; if we have to generate a dataset to train a deep model, we need many images like this and the time to create them will be huge. Even if rendering much simpler objects and using multiple GPUs in parallel, creating a big dataset to train a deep model would require a lot of resources and time.
Secondly, obtaining photorealistic images also requires designers to carefully prepare the scene to capture its realism. Obtaining a photorealistic image is not automatic at all and the time needed to model those scenes can be comparable to the time needed to manually label real data. Spending this time doing it is only worth if the labels are extremely hard to obtain in real images. The creation of this kind of scenes is not helpful otherwise.

In this work we propose to use a different approach called Domain Randomization, which has proven to be very successful in the generation of synthetic data for deep models. Instead of using highly realistic images to bridge the reality gap, this approach uses simpler representations with a wide variety of random changes in the

FIGURE 2.1: Example of a photorealistic scene used in [4] to compare rendering times in Blender with different hardware.

(A) Original Image (B) Photorealistic (C) Random

FIGURE 2.2: Example of two methods to bridge the reality gap. The first image is real data, which should be labeled manually. The second have been created using Blender Cycles and adding the same background to the image. Finally, the third approach is using domain randomization.

environment. Adding random variations in the scene, the network learns just the important and invariant features of the dataset.

Figure 2.2 shows a simple example to understand the idea behind domain randomization. The first image is the real data, which has to be manually labeled. The second one is a photorealistic approach, where the image tries to match the real scenario as much as possible. In this case we used Blender Cycles to obtain the object and we added the same background image as in the original one. Finally, the third approach is the domain randomization. This is much easier to obtain, since we do not really look for the exact same appearance. In this case, we randomize the background, the light conditions and the color of the image, so the trained network ignores those features.

The idea is that if we add enough variability to the synthetic images, the network will generalize better to real ones. [45, 47] are examples of the use of Domain Randomization to train deep models for different tasks. In this work, we will follow this approach using one of the simpler rendering engines available in Blender [5], called *Blender Render*. This engine will allow us to create fairly realistic images without much designing effort and little computational time.

2.1.2 Synthetic Data Generation

Since the work is focused on regressing the pose of single objects in an image, the generation method is fairly simple. In order to create images of a given object in multiple positions, we fix the camera location and rotation, and we move the object in a random way around the scene. The images are saved in Portable Network Graphics (PNG) format, so we are able to extract a mask if we want to get the segmentation image.

The generation of ground truth values is totally automatic using synthetic data. For each image, we record the following labels:

- **Segmentation Image**: For each image we create a transparency mask, defining which pixels contain the image and which ones are background. This is very useful when implementing the domain randomization, because we are able to easily change the background image.
- **Center**: For each image, we compute the center of the object as the centroid of all the pixels in the segmentation image. Therefore, using the transparency mask we can easily find the centroid by finding the mean position of all points in (x, y). This centroid value is normalized, so it is not affected by the image size.
- **Bounding Box**: For each image we also compute the bounding box of the object in the image. The bounding box is described as (x_i, y_i, x_l, y_l) where subindex i stands for initial point and subindex l stands for length.
- **Location**: We then compute the location (x, y, z) of the object with respect to the camera. Since we place the camera at location $(0, 0, 0)$, the position of the object is simply the same position of the object in Blender in world coordinates.
- **Rotation**: Finally, in order to capture the pose of the object, we use the rotation of the object with respect to an initial position. Since the Euler angles suffer from the well-studied problem of *gimbal lock* [2], we use Quaternions to express the rotation as (q_1, q_2, q_3, q_4).

Generation Pipeline

The location where the object is placed at each instance is decided randomly. However, in order to generate images in Blender efficiently, we need to ensure that the objects are never placed outside the camera view frame. In order words, we limit the position of the object within the *camera view frustum*. In order to do so, we first compute the vectors defining the camera view from the origin point onwards. Then we randomly pick a distance between the object and the camera frame. Thus, we project this distance to the projection vectors and compute the plane defining the camera view at that distance. Finally, we will randomly pick a point within that area and set it as the new location of the object. Figure 2.3 shows an illustration of the image generation scene.

In order to pick the rotation at each distance with a uniform distribution, we follow the approach described in [2]. Therefore, we can get a uniform distribution of

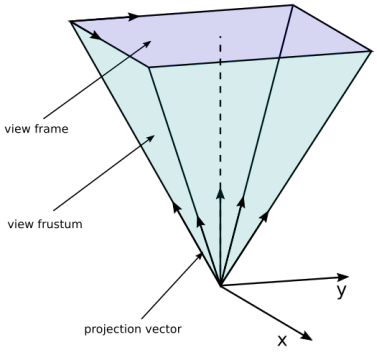


FIGURE 2.3: Illustration of the image generation scene.

quaternions with the following equation:

$$\begin{aligned} q = &\sqrt{1-u_1}\cdot\sin(2\pi u_2) + \sqrt{1-u_1}\cdot cos(2\pi u_2)i \\ &+ \sqrt{u_1}\cdot\sin(2\pi u_3)j + \sqrt{u_1}\cdot\cos(2\pi u_3)k \end{aligned} \tag{2.1}$$

One of the main benefits of creating synthetic images is that the process can be completely parallelized. In order to speed up the generation process we create multiple threads to work in parallel. In this work, we have only used one computer but, in real-world applications, this process could be distributed in multiple devices to create the synthetic dataset much faster.

Rendering Engine

There are many rendering engines available in Blender, from the well known OpenGL to third party engines that allow much more complex and realistic renderings. For this work we used different ones and, experimentally, we found the *Blender Render* to be the best option for our system. Blender Render is the internal engine of this software and, even though it cannot generate photorealistic images, it can obtain fairly accurate results. This engine produces much more realistic images than OpenGL and it is clearly much faster than any other complex rendering engine. For our purposes, this has the best trade-off between realism and acceptable computational time.

Figure 2.4 shows three examples with the three most usual rendering engines in Blender. As we can see, there is a huge difference in realism between OpenGL and the other two, specifically in the shading. In Blender, the OpenGL engine is very simple and it only allows environmental lighting (uniform lighting without bounces), so it is hard to create shadings with that engine[1]. With the other ones, two spotlights where placed in the scene to create the illumination and the shading. Notice that Blender Cycles only needs the two sources of light to create a natural and realistic

[1]Blender does not provide the exact computation time when rendering in OpenGL because it is considered real-time rendering. This time is the average time of generating the image 1000 times using the python API.

[1]OpenGL in Blender is only used as a display feature, but there are libraries in OpenGL that allow to create spotlights just like the other engines. These were not explored in this work.

(A) OpenGL[1] (B) Blender Render (C) Blender Cycles

FIGURE 2.4: Examples of three rendering results with the three most usual engines in Blender. All results are 1000×1000 images that were computed with an Intel® Core™ i7-7700HQ CPU (x8) at 2.80GHz and a NVIDIA GeForce GTX 1050. But only Blender Cycles uses GPU, the other two only use CPU.

image, because it computes a lot of bounces of the light forming a kind of environmental illumination. In Blender Render, we added this environmental illumination externally to obtain a better result as it is shown in Figure 2.5.

As we can observe in Figure 2.4, Blender Cycles clearly generates a more realistic image, but the computational time needed to create the image was also a lot higher. Blender Cycles was about 30 times slower than Blender Render in CPU. Notice that, even though we have not studied the resource usage in detail, doing the experiment, Blender Cycles entirely used all resources of the system, while Blender Render used less than a 30%. This difference is also very important, since the pipeline can use multiple threads to parallelize the process.

2.1.3 Domain Randomization

As it was explained in the first part of this section, this work focuses on domain randomization to reduce the reality gap between real images and synthetic ones. Therefore, instead of trying to produce realistic results to match the real images, we generate a simpler representation and randomize all the features we do not want the network to learn. With the above generation process, it is fairly easy to add domain randomization to the images. There are three basic modifications we apply to the images: background, light and object randomization.

Background Randomization

This is the simplest strategy to prevent the network from overfitting to background features during training. It simply consists in using absolutely random images as background. In our pipeline, this process is done after the rendering process. Since the image generated in Blender is in PNG format with transparent background, we can directly paste the object over a random image. As random images, we use random crops from the PascalVOC dataset [12] and from the DTD dataset[9]. In order to increase the variability even more, we also apply some random changes in brightness, contrast and saturation to both background and foreground images.

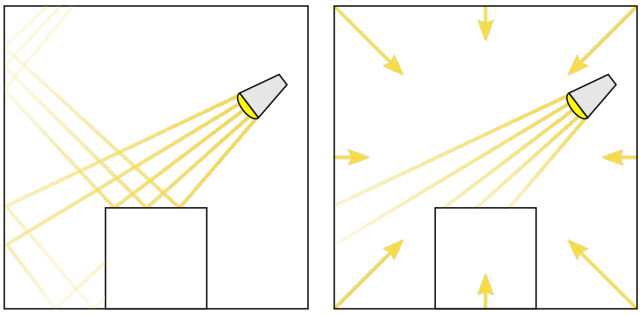


FIGURE 2.5: Simplification of the lighting used in Blender.

In manufacturing settings, the real background of the system in test time is usually known. If that is the case, those images are used in the synthetic generation process with a certain probability. However, note that even if the background is completely known, it is still important to use random images with a high probability and not just the real ones. Backgrounds usually have shades and reflections that are very hard to model with renders, so domain randomization will force the network to ignore those effects. In Chapter 4 we study the effect of either using simple backgrounds or random ones.

Light Randomization

One of the key elements that makes an image look real is the illumination. In the real world, the illumination comes from the direct rays produced by the different sources of light, but also from the indirect lighting produced by the bounces of those rays. Simulating those bounces is extremely hard and it is one of the reasons why photorealistic renders are so computationally expensive. In order to generate datasets relatively fast, we need to find a better way to produce that illumination.

The approach we follow with the illumination is much different. We want to create images that do not differ a lot from the real ones, but we want to simplify the illumination part. In order to do so, we limit the number of bounces and shades that a ray can produce, and we add an environmental illumination to produce the indirect lighting. This is much faster to produce than a complete simulation and it still generates results close to realistic. Figure 2.5 shows an illustration of the simplification.

A comparison between the complex lighting and this simplification can be seen in Figure 2.4 above. While the render obtained with Blender Cycles in Figure 2.4c did a complete simulation of the lighting, Figure 2.4b was obtained with the Blender Render combining simpler direct light and environmental light. As we can see, even though Blender Cycles generated a more realistic image, the other one produced a fairly good approximation. Notice that Blender Render also have the option to create a more complex scene like Cycles, but it also increases a lot the computational time.

Using this simplification we can speed up the rendering process, but we need to ensure that there is enough variation in the illumination. In order to achieve good results with domain randomization, we need to ensure that there are as many combinations of lights as possible. This is achieved by placing many spotlights uniformly distributed over the scene and randomly activating some of them. During the generation process, the illumination comes from a random number of spotlights, with

random light intensities. Of course, this lighting does not affect the background, because there is no background in the rendered scene. However, using the background randomization described above we overcome this limitation.

Object Randomization

Finally, we need to add some variations to the object itself. The color, materials and shape of real objects are very hard to match. Even spending a lot of time modeling real objects, in real manufacturing settings, there are always deviations between different instances. The idea of object randomization is to apply small variations to this properties to make the network more robust to those differences. This is achieved by using Gaussian distributions for the color and shape values. Therefore, instead of a unique color, the rendered color is a random pick for each RGB color selected from a Gaussian distribution with mean equal to the nominal value. Similarly, we apply small deformations to the shape, by varying the 3D scale vector, also using a Gaussian distribution for each element.

In the case of objects with image textures like serigraphs, the same variations can be applied to the colors of the texture. In addition, when rendering objects with those kind of textures, we also vary the relative position of the image with respect to the object. Therefore, in the case of a serigraph, the image would appear slightly shifted on each instance.

2.2 Dataset Creation

This work contributes with a new dataset and a framework for manually annotating new data for 6D pose regression tasks. First, this section describes the program called *6D-Labeler*, which can be used to quickly create new annotated datasets. Secondly, this section describes the dataset created using this program that will be used to test our system in Chapter 4.

Even though the available datasets in the literature are perfectly valid for evaluating deep models, having a way to create new datasets provides much more flexibility. The idea behind this framework is to generate labeled images with the same conditions that we would find in real-world industrial applications. The proposed framework allows us to prepare datasets with the exact conditions and objects that we want to test.

Most of the available datasets described above have either cluttered scenes, very complex backgrounds, poor light conditions or even semi-occluded objects. We propose a much more controlled scenario, where there is only one object in the scene, the background is fixed and the light conditions are good. This is usually the case in many industrial automation applications.

2.2.1 6D-Labeler

6D-Labeler is a program developed in Python 3 and OpenCV [7] that can be used to create annotated datasets for 6D Pose Regression tasks for RGB images. This is accomplished by manually clicking known points of the object in each image and

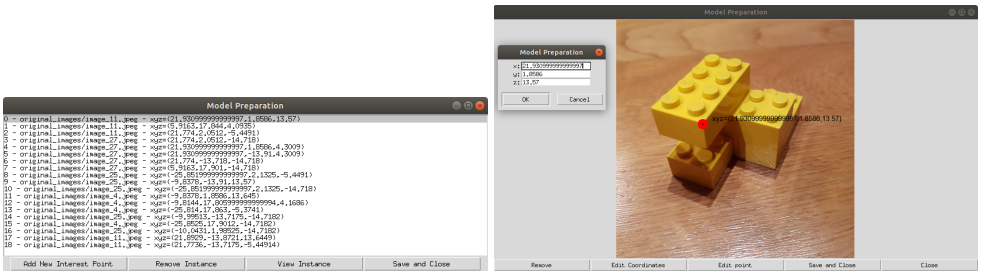


FIGURE 2.6: Screenshots of the Model Preparation Interface. The first image shows the main menu, where we can add or edit the points and images describing them. The second image shows the interface for setting the 3D coordinates for one point.

applying a PnP algorithm to regress the pose. This program offers an easy interface for defining interest points for objects, finding those points in RGB images, calibrating the camera and generating the final dataset.
The main advantage of this framework is that it can regress the pose of an object without any other reference than the object itself and without the need of depth information. This means that the final image does not need calibration markers (like in the LINEMOD [21] or the T-LESS [22] datasets) and it can be obtained with a simple camera.

The process to create a new dataset is divided into two steps: the model preparation and the manual annotation. The first part prepares a new model to be labeled. This means the user have to define the interest points of the model that will later be used to label the images. The interface allows the user to input some images where the interest points are visible and use those images to define the set of interest points.
The second part is an interface that allows the user to manually click on each of the interest points visible on the image. This interactive interface allows the user to quickly find the defined interest points and check the final result.

Model Preparation

In this part, we can prepare a new model to be labeled. With an intuitive interface, we define which is the object of the dataset and which are the interest points that will be used to regress the pose in the manual annotation step. Figure 2.6 shows two screenshots of this interface: the main menu and the editing of interest points.

In this step we need to specify as many interest points as possible to simplify the manual annotation in the following part. For each point, the interface asks for the 3D coordinates of that point in the model, which can be easily obtained using a 3D visualizer like Blender [5].

Manual Annotation

The second part of the program is an interface to manually label images using the model defined previously. By relating 2D points in the images to 3D coordinates in the model, we can easily regress the location and rotation with a PnP algorithm. The algorithm used to compute the regression is an iterative method based on *Levenberg-Marquardt* optimization, already implemented in the OpenCV library [7].

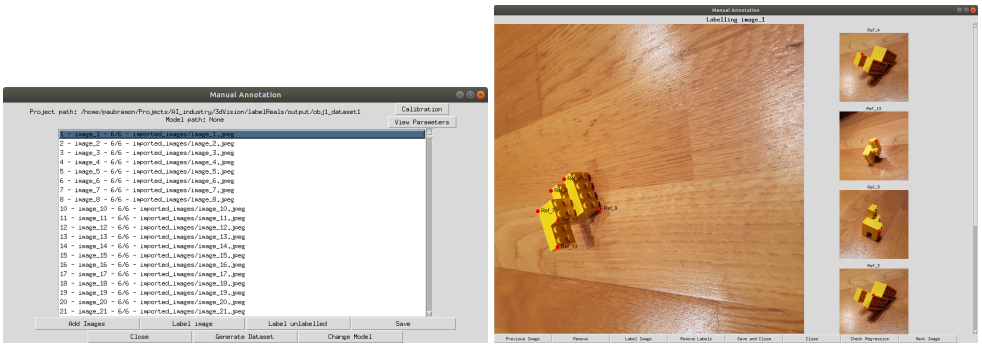


FIGURE 2.7: Screenshots of the Manual Annotation Interface. The first image is the menu where we can manage the images added to the dataset, label new images, calibrate the camera or finally creating the dataset. The second image is an example of manual annotation, where we can relate 2D points in the image with 3D coordinates.

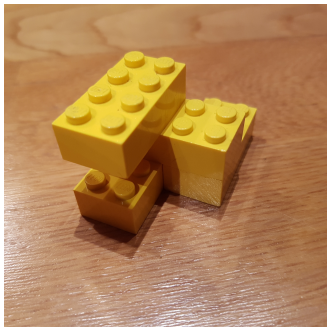
FIGURE 2.8: Example of two verification images generated by the 6D-Labeler.

Defining just four points per image should be enough to correctly predict the pose without ambiguity. However, since the manual annotation might not be exact and the points defined might be coplanar, the algorithm might fail using only four points. In practice, it is better to set more than four points per image to ensure the pose is correctly predicted.

Figure 2.7 shows some screenshots of the interface. The first image is the main menu, where we can manage the images being labeled, save the work done and generate the final dataset. The second image is the interface where we can relate 2D points in the RGB image with 3D coordinates of the model.

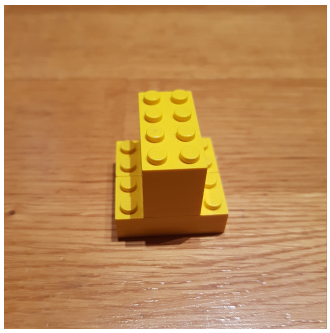
After every annotated image, the interface shows a verification image to check the predicted pose. In this check, the original image is blended with a render of the 3D model in the predicted pose. Therefore, if the manual annotation was incorrect, we could see it in the verification image and correct it. Figure 2.8 shows two examples of verification images.

Finally, in the main panel, there is a button to calibrate the camera. Selecting images of a calibration grid, the interface will be able to compute the camera matrix and the distortion coefficients using the OpenCV library [7]. These parameters are necessary to precisely regress the pose with the PnP algorithm.



(A) Toy

(B) Box



(C) Symmetric

FIGURE 2.9: Dataset objects.

2.2.2 6D-Single Object Dataset

This work contributes with a new dataset called 6D-Single Object Dataset (6DSO), created using the 6D-Labeler. This dataset contains three different objects with a very simple background and good light conditions. Each image contains a single object with no occlusions. This dataset intends to simulate a common industrial automation environment, where the light conditions are good, the background is simple and nearly static, and the system can ensure there is only one object per image.

So far, the dataset only contains three different objects, but it could be extended in the future. The three objects have different characteristics that tries to emulate the problems that the system we propose could face in real industrial applications:

- **Toy**: this object has a complex shape, where two photos of the same object might be completely different depending on the pose. This shape has a lot of small details that produces a lot of different and complex shades. It is interesting because it is similar to what we could find in manufacturing settings. Figure 2.9a shows an image of this object.
- **Box**: this object is a regular box with a concrete serigraph. The complexity of this class is that the real object and the 3D model are not exactly the same. The box is deformable, therefore it is impossible to have an exact model of it. Furthermore, the serigraph may not be exactly at the same place as in the 3D model. We want the network to absorb all these differences. Figure 2.9b shows an image of this object.
- **Symmetric**: this object is very similar to the Toy, but it have an axis of symmetry. The reason to include this object in the dataset is to see the effect of symmetries in the training of the network. Figure 2.9c shows an image of this object.

The dataset contains 1000 images per object in various locations and rotations. The illumination changes so the images contains many variations in the shading and light intensity.

Chapter 3

Network

This chapter defines the structure and details of the proposed network for the 6D pose regression task. In the first section, we give an overview of the network and its multi-task structure. In the second one, we analyze in detail the different parts of the network and the way they are trained. In the third section, we propose the use of a new loss function for the pose regression task, that reduces the errors produced by view ambiguities and symmetries. Finally, in the fourth section, we provide some details of the network implementation and the data augmentation process.

3.1 Overview of the Network

As it was already stated, the system is intended to work in industrial environments and the task is to regress the location and rotation of isolated objects. Consequently, in this work, it is assumed that the input image only contains one object of interest at a time. The defined network is trained to produce a prediction of the 6D pose for a unique object and ignore any other element appearing in it. The network could be extended in the future to be able to regress multiple instances of the same object class or even to detect different classes. However, for this work, the network is limited to one class and one instance at every input image. In fact, this is a very common scenario in many manufacturing configurations, where the camera is carefully set to take photos of isolated incoming products.

The input of our network is an RGB image with fixed size, containing one object to be regressed. This size can be defined depending on the quality of the images or the relative size of the object within the scene. On the one hand, making the image bigger gives the network more details of the object to regress. But on the other hand, increasing the size of the image also increases the number of parameters of the network, increasing the complexity of the optimization process, the inference time and the memory size.

The network regresses the location of the object with respect to the camera and the rotation of the model with respect to an original orientation. It is assumed that the object's 3D model is available as a matrix of 3D point locations $\mathbf{M}_{P,3}$, where P is the number of mesh points and each point is defined with three coordinates (x, y, z). On the one hand, the predicted location vector $\mathbf{t}$ is defined as the position of the origin of the model coordinate system with respect to the camera location. On the other hand, the predicted rotation vector $\mathbf{q}$ is defined as the rotation of the model coordinate system from its initial orientation. The prediction for the rotation is expressed as the quaternion applied to the model coordinates to obtain the pose in the input image.

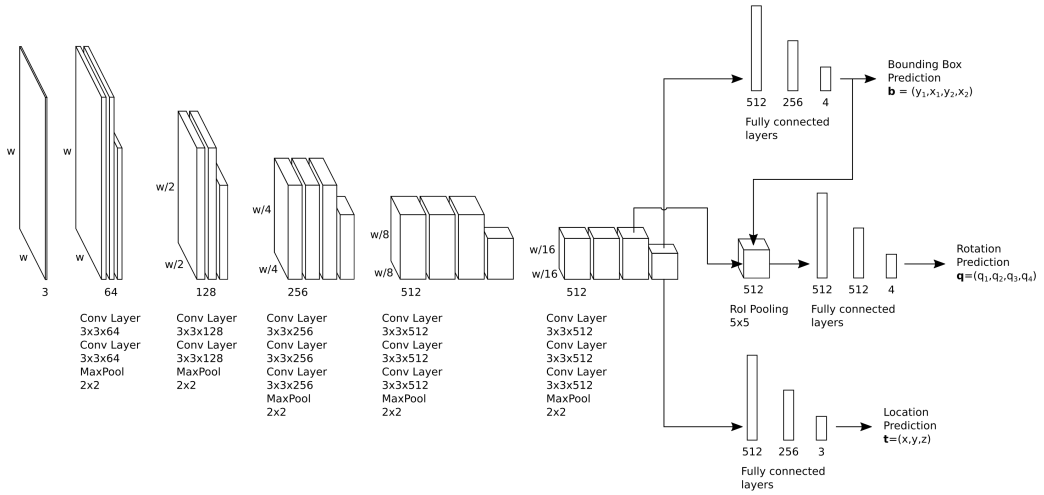


FIGURE 3.1: Architecture of the proposed network.

Figure 3.1 illustrates the architecture of the proposed network. As it can be observed, the network is composed of a main backbone, which extracts all the necessary features from the image, and three branches to perform three different tasks.
The first task is the bounding box prediction, locating within the image the 2D coordinates of a box tightly containing the object to regress. Even though this is not the aim of the network, this information is useful to guide the optimization process.
The second branch finds the rotation vector $\mathbf{q}$ defined above, which defines the quaternion that tells how the mesh should be rotated to match the input image. Notice that this task does not use all the features extracted by the backbone, but only the ones contained within the bounding box. This is done by means of a RoI pooling mask, which crops the tensor coming from the backbone to the bounding box area predicted above.
Finally, the third branch finds the location of the object with respect to the camera $\mathbf{t}$. This task uses again all the features extracted by the backbone.

In the architecture shown in Figure 3.1, it can be noticed that the backbone of the network follows an structure similar to the VGG16 network [42], which uses 13 convolutional layers and 4 max pooling layers. We add another pooling layer to reduce the size of the last layer before the bounding box and location predictions. Using this common structure we can use available pre-trained weights from other works to get a faster training.

Concretely, this network uses the following two main concepts to properly regress the 6D pose:

- Firstly, the use of a multi-task neural network to separate as much as possible the different sub-processes necessary for the pose regression. The reason to use a multitask network is that the optimization process can be guided more precisely with many different loss functions. In the literature, there are several works using similar architectures [50, 11] and this work focuses on this approach.
- Secondly, the use of a new loss function that we call *Projection Loss*, which aims to reduce the errors produced by view ambiguities in some objects. In the

literature, other works propose different methods to deal with those errors, from treating the problem as a classification task [26] to adding other kind of losses [50]. The *Projection Loss* is inspired by some refinement methods [31, 36], which compares the foreground masks of the predicted and ground truth poses to correct an initial prediction. We propose to integrate this comparison directly into the multi-task neural network and train the system end-to-end.

3.2 Multi-task network

As it can be observed in Figure 3.1, the proposed network performs three different tasks separately. The idea is simple, the first branch is intended to locate the object within the image, the second branch computes the rotation vector $\mathbf{q}$ and the third regresses the translation vector $\mathbf{t}$. The loss of the network is the weighted sum of the independent losses of the three tasks, a new loss we define as *Projection Loss* and a regularization loss. Therefore, the total loss to minimize can be computed as follows:

$$\mathcal{L}_{total} = \lambda_b \mathcal{L}_b + \lambda_q \mathcal{L}_q + \lambda_t \mathcal{L}_t + \lambda_{PL} \mathcal{L}_{PL} + \lambda_{reg} \mathcal{L}_{reg} \tag{3.1}$$

where $\mathcal{L}_b$, $\mathcal{L}_q$ and $\mathcal{L}_t$ are the losses of the bounding box, rotation and translation task respectively. The $\mathcal{L}_{PL}$ is the new *Projection Loss* defined in the following section and the $\mathcal{L}_{reg}$ is an L2 regularization loss of the fully connected layers. The λ parameters weight the contribution of each loss to the final loss and they are selected such that all losses contribute similarly.

3.2.1 Bounding Box Prediction

Accurately regressing the bounding box of an object is not the purpose of the network itself, but this task is very useful for the other two. Firstly, because the bounding box is used in the rotation prediction branch to select only the features from within the area of the object. Secondly, because adding the bounding box loss to the training process also helps to train the feature extractor of the network faster. Finding where an object is in the image is an implicit task when regressing the location and rotation, so adding this separate loss helps the training process.

This part of the network is pretty straightforward; the network uses all features from the backbone as the input to three fully connected layers. In order to do that, the last layer is flattened and connected to a fully connected layer with 512 neurons with Rectified Linear Unit (ReLU) activations. Using another hidden layer of 256 fully connected neurons with ReLU activations and a linear output layer of four units, we obtain the bounding box prediction. This last four neurons correspond to the bounding box vector $\mathbf{b}$. This vector is formed by two normalized points describing the top-left corner (y_1, x_1) and bottom-right corner (y_2, x_2) of the bounding box (y_1, x_1, y_2, x_2).

The bounding box loss is then simply computed as the Euclidean Distance:

$$\mathcal{L}_b = \|\tilde{\mathbf{b}} - \mathbf{b}\| \tag{3.2}$$

where $\mathbf{b}$ and $\tilde{\mathbf{b}}$ are the ground truth and estimated bounding box vectors.

3.2.2 Rotation Prediction

The second branch of our network is intended to predict the rotation of the object appearing in the image. As we already stated, we define this as a quaternion that describes how the 3D model should be rotated to obtain the pose shown in the image.

Intuitively, it is clear that the information needed to compute this quaternion is only contained within the bounding box area. So all features from any other part of the image are obviously not useful and therefore discarded. This is why we used a RoI pooling layer in this branch, which only picks the features within the bounding box area and discard all the rest. Then, the network resizes the crops obtained from the RoI pooling layer to a fixed size and flattens the neurons to be inputted into a fully connected layer with 512 units and ReLU activations. Finally we use another fully connected layer with 512 units and ReLU activations, and an output linear layer with 4 units. The four neurons in the output layer form the quaternion vector $\mathbf{q}$.

In order to compute the loss for this task, we cannot naively use the euclidean distance like in the bounding box one. The first problem of comparing quaternions using a distance metric is that we need to take into account that two quaternions $\mathbf{q}$ and $-\mathbf{q}$ represent the same orientation. This is very important in an optimization process, because we need to be consistent in the computation of the loss; if the image visually contains the same object with the same orientation as the prediction, the loss must be zero. In order to overcome the problem, we could restrict the quaternions to be always positive. We could ensure the uniqueness of rotation for each quaternion (q_1, q_2, q_3, q_4), by forcing the first element q_1 to be always positive [30] (considering that q_1 will never be exactly 0).
However, there is another problem in using a simple distance metric to compare two rotations (either using quaternions or Euler angles). As we already stated, the computation of the rotation loss have to be always consistent: similar poses should produce a lower loss and different poses should produce a higher one. Since we defined the orientation of the object as the rotation with respect to a rest pose, the loss would not be consistent using directly a distance metric. Two similar poses can be reached with very different rotation vectors and, consequently, produce a high loss when it should not. For example, the two rotation vectors $\mathbf{r_1} = (179^o, 0^o, 0^o)$ (with an approximate quaternion $\mathbf{q_1} = (0.009, 1, 0, 0)$) and $\mathbf{r_2} = (-179^o, 0^o, 0^o)$ (with an approximate quaternion $\mathbf{q_2} = (0.001, -1, 0, 0)$), would produce a high loss but the poses would be in fact very close (just 2^o in x).

Since directly comparing the rotation vectors does not seem a good solution, we followed a completely different approach. We experimentally found that an approach similar to [50] produces a much more consistent loss and it makes the training much more efficient. We define the rotation loss as the mean distance between the points of the 3D model in the predicted and ground truth poses:

$$\mathcal{L}_r = \frac{1}{P} \sum_{\mathbf{x} \in \mathcal{M}} \|\mathbf{R}\mathbf{x} - \tilde{\mathbf{R}}\mathbf{x}\| \tag{3.3}$$

where $\mathcal{M}$ is the set of points of the 3D model (i.e. all the rows in the $\mathbf{M}_{P,3}$ matrix) and P is the total number of points in the set. $\mathbf{R}$ and $\tilde{\mathbf{R}}$ are the ground truth and predicted rotation matrices respectively, which can be easily obtained from its corresponding quaternions as follows:

$$\mathbf{R} = \begin{bmatrix} 1-2s(q_3^2+q_4^2) & 2s(q_2q_3-q_4q_1) & 2s(q_2q_4+q_3q_1) \\ 2s(q_2q_3+q_4q_1) & 1-2s(q_2^2+q_4^2) & 2s(q_3q_4-q_2q_1) \\ 2s(q_2q_4-q_3q_1) & 2s(q_3q_4+q_2q_1) & 1-2s(q_2^2+q_3^2) \end{bmatrix} \tag{3.4}$$

where the quaternion has the form $\mathbf{q} = (q_1, q_2, q_3, q_4)$ and the parameter s is the inverse of its square module $\|\mathbf{q}\|^2$.

Intuitively, this loss seems much more consistent than a distance metric; the farther the prediction from the ground truth, the higher the loss is. Notice that in practice, for computational reasons, we will use just a subset of points from the 3D model to compute this loss, because using all the points does not add much more information.

3.2.3 Location Prediction

For this task we use the same approach used with the bounding box. This branch uses all the features from the last layer of the backbone with three fully connected layers to regress the location. The last layer of the backbone is flattened and inputted to a fully connected layer with 512 neurons and ReLU activations. Then, another fully connected layer with 256 units with ReLU activations is used and an output linear layer with 3 outputs produces the final translation vector $\mathbf{t}$. As we already stated, this will directly be the location of the object with respect to the camera coordinates.

The computation of the location loss is pretty straightforward. Just like we did with the bounding box loss, we will use the euclidean distance between the predicted and ground truth vectors. Therefore, the location loss will be computed as follows:

$$\mathcal{L}_t = \|\tilde{\mathbf{t}} - \mathbf{t}\| \tag{3.5}$$

where $\mathbf{t}$ and $\tilde{\mathbf{t}}$ are the ground truth and predicted location vectors (x, y, z).

3.3 Projection Loss

Even though the network defined in the previous section would already obtain good results in the 6D pose regression task, this section introduces the *Projection Loss* to improve a bit more the results. We experimentally found that the network is not able to properly learn some scenarios with ambiguous views, because, for these situations, the optimization process gets stuck in very poor local minima. The aim of this loss is to address this problem by adding a loss that compares the projection of the predicted and ground truth poses.

3.3.1 Dealing with view ambiguity and symmetries

The 6D pose regression task sometimes has to deal with view ambiguities. Many objects have some views from where it is impossible to determine its pose. Figure 3.2 shows a very simple example of an object that have an ambiguous view. As we can observe in the figure, if we look at the image from point A, it it impossible

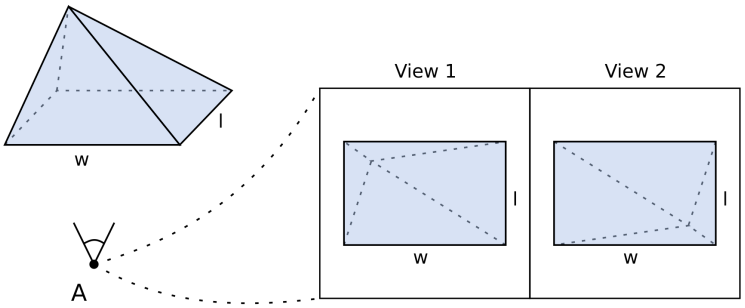


FIGURE 3.2: Illustration of the view ambiguity problem.

to deterministically regress the pose, because the useful information is simply not visible. Since the network only uses a single RGB image, the only solution in these cases is to randomly pick one of the valid options (either View 1 or View 2 in the figure example).

Ambiguities like the one shown in Figure 3.2 are impossible to solve with single RGB images, but they are just extreme cases. For example, the object in that figure just have an ambiguous view when the photo is taken from point A. In most of the cases, the view will show some other part of the object that will break the ambiguity, making the problem deterministic again. Furthermore, the problem is simply unsolvable with the information available, so there is no other solution than making a random guess.

However, we experimentally found that these ambiguities hugely affect the optimization process. During training, the network is learning what features determine the pose of an object and how these features are related spatially. When the network gets an image like the one shown in Figure 3.2, it has to make a prediction that will be evaluated with the losses defined previously. The problem is that, only with the current losses, the result is not consistent and for the same input and output images, the loss can be different.
In the best case scenario, where we can perfectly predict the pose, the loss after an ambiguous image like the example will still be high with a 50% chance. This means that, even though the features responsible for the prediction during the forward pass were correct, with a 50% chance, the gradients during the backward pass will try to modify them. This effect will not only slow down the training process, but it will also lead to bad results. With the losses defined previously, the network might end up learning to predict a pose in between the two valid options, because it finds that the minimum error is at that middle point. In the evaluation chapter we will see this effect in some real examples.

The Projection Loss aims to solve this problem by comparing the projection of the 3D model over the camera frame for the prediction and the ground truth poses. Intuitively, this loss seeks to match the silhouette of the predicted pose with the silhouette of the object in the image. The hypothesis is that the loss will improve the learning by forcing the network to pick one of the valid views in an ambiguous case and eliminate the local minimum in between. For example, in the scenario described in Figure 3.2, this Projection Loss would force the network to predict one of the ambiguous views, helping the optimization process.

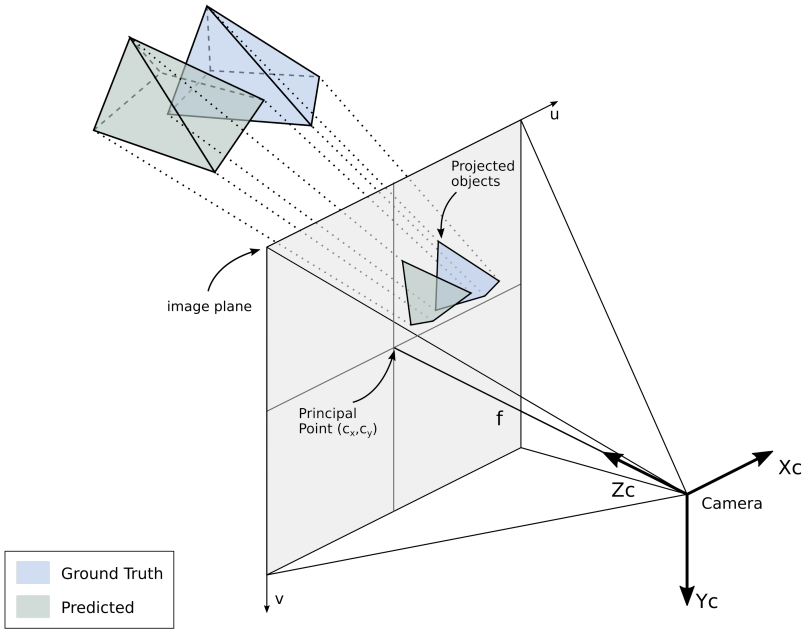


FIGURE 3.3: Illustration of the Projection Loss calculation using the Pinhole Camera Model.

Symmetric objects are just an extreme case of ambiguous views. All images in a symmetric object have two or more possible poses that looks exactly the same. If the object is perfectly symmetric, these ambiguities can be eliminated by restricting the poses used for training to the poses of just the non-symmetric part of the object. However, if the objects are not exactly symmetric, this trick cannot be applied. With the Projection Loss we aim to solve this issue in a generic way, for perfectly symmetric objects and for almost symmetric ones.

3.3.2 Definition of the Projection Loss

We define the Projection Loss as a measure that compares the projections to the camera image plane of the pose prediction and the ground truth values. The measure to evaluate the similarity of the two projections is the Intersection Over Union (IoU). Transforming this measure into a loss value, we can compare whether the two silhouettes match or not. This is more or less the same as checking how similar the object in the predicted pose would look from the point of view of the camera. Figure 3.3 shows an illustration of the projection loss calculation, where we use the basic pinhole camera model [19]. Intuitively, this loss is similar to the process we humans do when trying to figure out the position of an object; we can only compare the poses we predict with what we actually see in the image.

In order to use this loss to train neural networks, we need to define not only how the loss is computed in the forward pass, but also how it behaves in the backward pass. The forward pass, is the obvious calculation, we define the value of the loss given the prediction and the ground truth. But from computation of the loss itself, we need to define how this loss affects the update of the network's weights. In other words, we need to define how the gradients will flow backwards during the backpropagation phase.

Forward Pass

The forward pass defines how the loss is computed using the predicted pose and the ground truth pose. The first step is to compute the projection of the two poses to a theoretical camera frame. Using the basic pinhole camera model [19, 8] (as shown in Figure 3.3), the projection of every point of the 3D model can be computed. The pinhole model allows a very simple mapping of points from Euclidean 3-space $\mathbb{R}^3$ to Euclidean 2-space $\mathbb{R}^2$. In this model, each 3D point is mapped to the image plane where the line joining the 3D point and the center of the camera (the pinhole) meets the image plane. Using the pinhole camera model and considering the camera in the origin of the coordinate system pointing straight down the z-axis, the 2D projections of the 3D model points of an object at a given rotation and location can be calculated as follows:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.6}$$

which can be expressed as:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{K}[\mathbf{R}|\mathbf{t}] \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3.7}$$

where $\mathbf{R}$ is the rotation matrix, $\mathbf{t}$ is the translation vector and $\mathbf{K}$ is the so called *camera calibration matrix* or *matrix of intrinsic parameters*. The values u and v are the coordinates of the projected point in the image plane (see the illustration in Figure 3.3) and s is the scale factor. With this equation, we can easily find, for each point of the 3D model (x, y, z), the projection in the image plane of a simplistic pinhole model.

In the equations above, $\mathbf{K}$ is defined by the intrinsic parameters: f_x, f_y, c_x and c_y. (c_x,c_y) is the principal point in pixels, which corresponds to the point where the line from the camera center perpendicularly meets the image plane. Ideally, this will be exactly the image center, but in real cameras there is usually some displacement. The two other parameters (f_x, f_y) are the *focal length* (f in Figure 3.3) in pixels. The reason for using two different focal lengths is that the pixels in real cameras are usually rectangular rather than square, so the distance of f in pixels is usually different for the two dimensions.

However, for the Projection Loss, obtaining a precise projection using the parameters of a real camera is not very important. This loss is comparing two projections (the predicted and the ground truth), so differences between a real camera and an ideal one will be the same for the two. Therefore, this loss will simply use an ideal matrix of intrinsic parameters with $f_x = f_y = 1$ and $c_x = c_y = 1/2$, to obtain the normalized 2D positions (u, v). There will be obviously differences between the projected image in the Projection Loss and the real one in the input image, but the differences will be equally present in both the predicted projection and the ground truth one.
Furthermore, notice that in real scenarios, the pinhole model is also not very accurate. Instead of a single point, real cameras uses lenses to gather more light, which

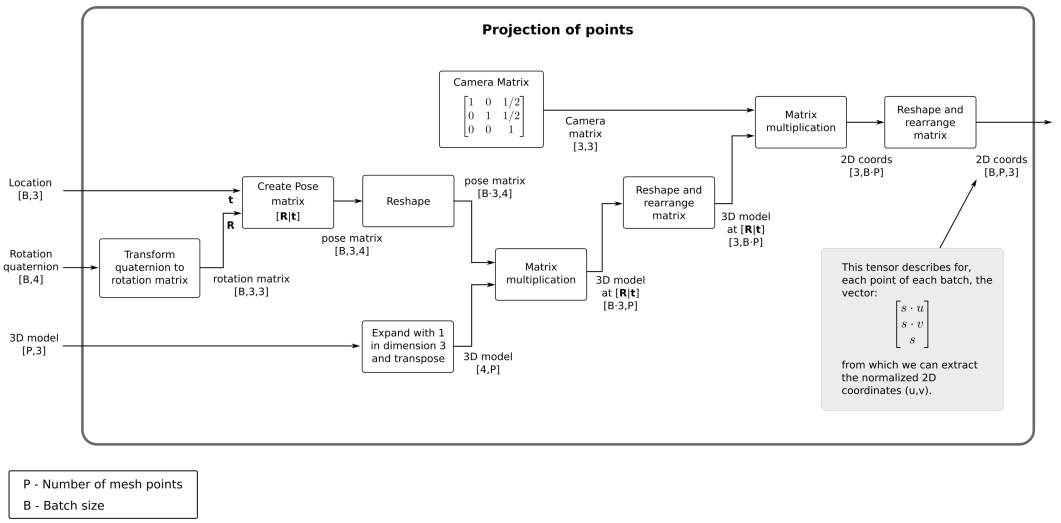


FIGURE 3.4: Illustration of the implementation of the first part of the loss as a batch operation. This part finds for each point of the 3D model, its projection on to the camera image plane.

makes the pinhole model inaccurate. Real lenses usually add distortion (radial and tangential distortion [8]), so the correspondences of 3D points in the 2D plane are not direct projections as in the pinhole model. Therefore, using the real camera matrix would not lead to precise results either, because the model is already a simplification. For the Projection Loss, the simplest model with ideal parameters will be enough to compare the two projections.

This first part of the loss was directly implemented in TensorFlow [1]. In order to do an efficient training, the operations described above where implemented for batches instead of single images. Figure 3.4 shows an illustration of the implementation as a batch operation. In the image, it can be seen that the tensors have to be rearranged and reshaped in order to perform all the necessary matrix multiplications for multiple images at the same time.

First, the points of the 3D model are transformed to the correct position in the camera coordinates. In order to perform this operation, the 3D tensor containing the pose matrix $[\mathbf{R}|\mathbf{t}]$ for each batch is transformed to a single matrix by concatenating one batch after another. This allow us to perform the matrix multiplication between the pose matrix and the set of point coordinates, resulting in the new coordinates for each point and batch.
Secondly, the points of the 3D model in the transformed position are projected to the camera image plane. The projection is performed by multiplying the ideal camera matrix described above with each point of the 3D model. In order to do this operation, we need to use the result from the first multiplication with shape $[B \cdot 3, P]$ (being B the number of batches and P the number of points) and rearrange the matrix such that each column contains a 3D point (transforming the matrix to a shape $[3, B \cdot P]$). The result of this multiplication is the left part in Equation 3.6, from which we can easily extract the normalized 2D coordinates (u, v).

The implementation shown in Figure 3.4 is the first part of the Projection Loss computation and it will be used for both the predicted and ground truth values. The second part uses the obtained projected points to generate the two masks and compare them with the IoU metric. This part is a bit more complex and it is harder to

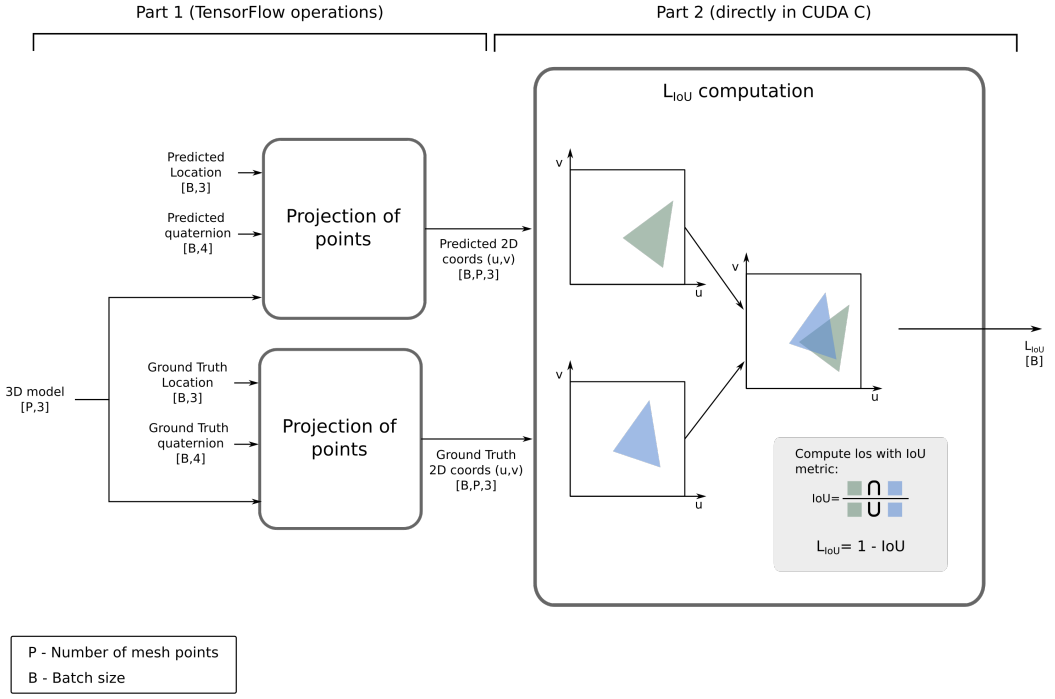


FIGURE 3.5: Graph of the Projection Loss implementation. The graph has two main parts, the first part is directly implemented in TensorFlow with the operations described in Figure 3.4 and the second part is implemented in CUDA C[34]

implement directly in TensorFlow. As it is described in the following section, this second step of the loss is directly implemented in CUDA C[34] and integrated into the TensorFlow implementation. Figure 3.5 shows the complete picture of the Projection Loss computation.

As it can be seen in Figure 3.5, the second part takes the 2D coordinates and, defining an arbitrary resolution, it creates two binary masks corresponding to the silhouette of the two poses. The input tensor is a $[B, P, 3]$ tensor, where for each batch element and point, a vector $(u, v, 1)$ can be extracted. Notice that the third dimension is used to normalize the 2D coordinates, because from this dimension we can extract s in Equation 3.6.
Being V the set of pixels of the predicted images, $X \in \{0,1\}^V$ the predicted binary mask and $Y \in \{0,1\}^V$ the ground truth binary mask, the IoU measure can be computed as follows:

$$IoU = \frac{I(X,Y)}{U(X,Y)} \tag{3.8}$$

where $I(X,Y)$ and $U(X,Y)$ are the intersection and union of the two masks, and can be written as:

$$I(X,Y) = \sum_{v \in V} X_v \cdot Y_v \tag{3.9}$$

$$U(X,Y) = \sum_{v \in V} X_v + Y_v - X_v \cdot Y_v \tag{3.10}$$

Using the IoU measure, the Projection Loss is computed as:

$$L_{PL} = 1 - IoU \tag{3.11}$$

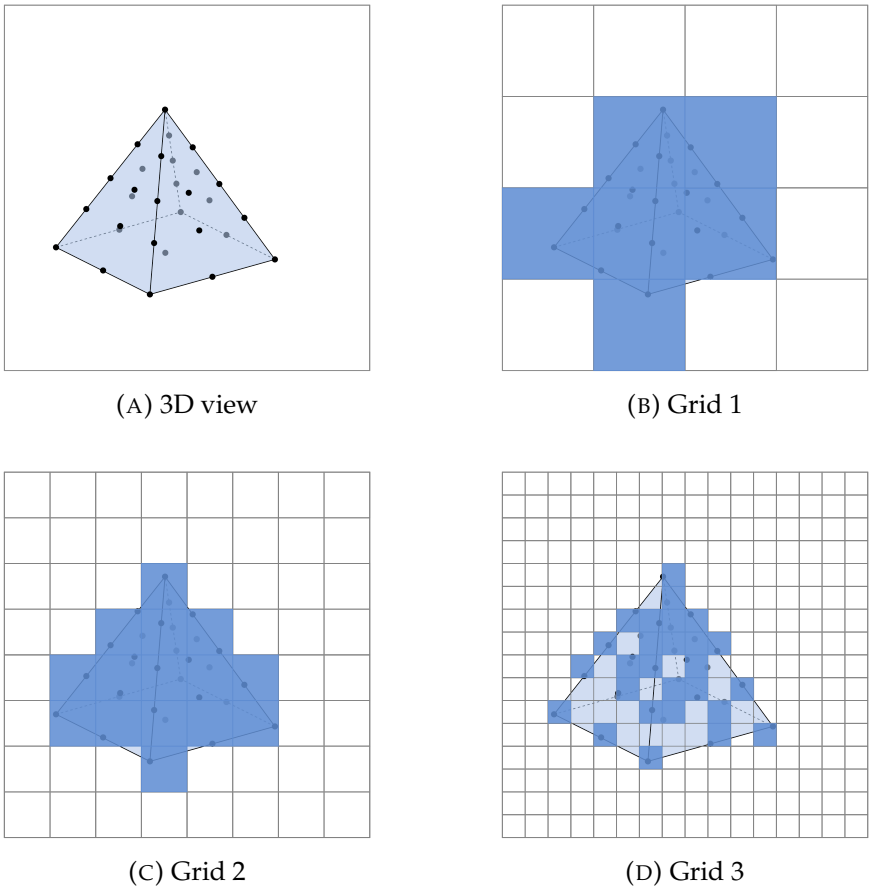
being zero when the two projections perfectly match and one when they do not intersect at all.

The main problem of this loss is that we are using a discrete number of points to create the projections (a finite number of points from the 3D model), so the projections might not be continuous. In order to use the IoU to compare the two silhouettes, the objects should be represented as a compact and continuous area in the image plane, rather than a scattered set of projected points. On the contrary, small variations between the two projections could produce big projection losses, because the scattered points would not be exactly at the same pixel.

To reduce this problem, there are basically two different options. On the one hand, we could use a huge number of points to represent the 3D model[1]. Using an enormous amount of points would clearly avoid the scattering problem, but it would be computationally expensive.
On the other hand, we can reduce the number of pixels in the projected image. Making the projected image smaller, the 2D points would be close enough to form a continuous area. However, making the image too small would create undefined masks that would not provide relevant information for the pose prediction. Figure 3.6 shows an illustration of this problem, where in D the resolution is too high and in B the resolution is too low.
In practice, we need to find a balance between obtaining defined images and being computationally efficient. Increasing the number of points in the model will allow us to use higher resolutions in the projected image and, consequently, have better defined masks. Reducing the size of the projected image will allow us to reduce the number of points of the model, making the computation of the loss faster.

Finally, notice that another problem affecting the projection loss is the use of 3D models with unevenly distributed point clouds. Usually, 3D models have higher concentrations of points in some parts of the object than in others. This is a problem because it can lead to some parts of the object to be poorly represented in the projected image. This problem can be solved by using a modified version of the 3D model with uniformly distributed points to compute the projection loss. This modified point cloud can be easily created with any three-dimensional computer graphics editor and it only have to preserve the main traits of the original model and not the small details.

[1] With any three-dimensional computer graphics editor like Blender [5], it is fairly simple to increase the number of points in a 3D model



(A) 3D view (B) Grid 1

(C) Grid 2 (D) Grid 3

FIGURE 3.6: Illustration of the scattering problem with the Projection Loss. While the projection in *B* does not really look like the silhouette of the original image, the projection in *D* is too scattered and would not work for the IoU measure.

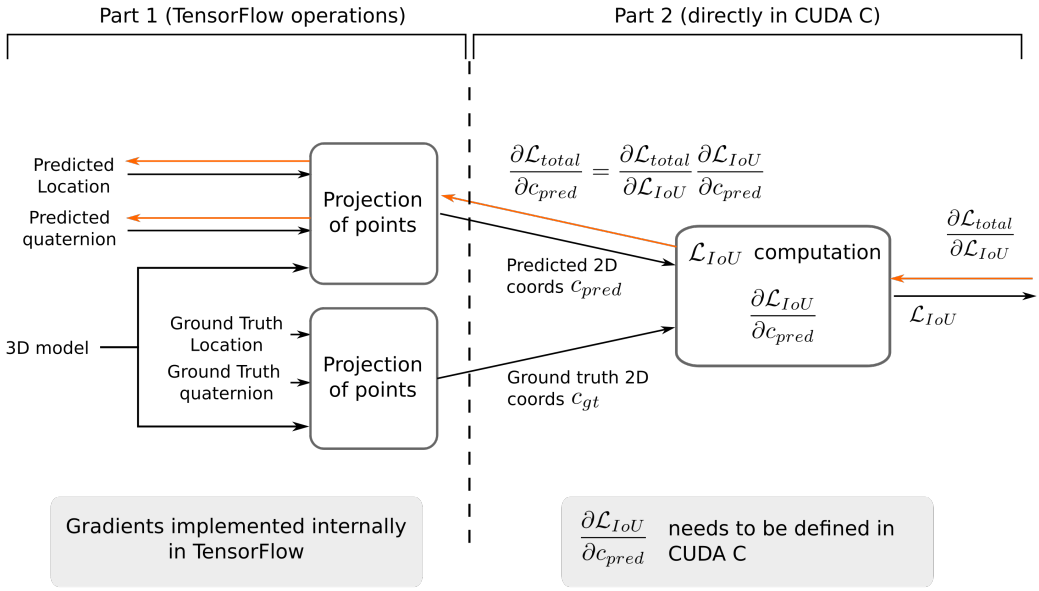


FIGURE 3.7: Illustration of the gradient propagation in the Projection Loss layer. The first part of the network are basic operations in TensorFlow, so the libraries have the backward pass already defined. The second part needs to be implemented manually, applying the chain rule to backpropagate the gradient coming from the following operations.

Backward Pass

As any other layer of the network, the Projection Loss also has to be defined for the backward pass. We need to define how the gradients coming from $\mathcal{L}_{total}$ will flow through the layers we just defined. In other words, we need to define the local gradients of the gates defined in Figure 3.5.
The first part of the loss (first half in Figure 3.5) is simply a combination of matrix multiplications and tensor manipulations. All these operations are differentiable and, therefore, the local gradients are already implemented in the TensorFlow libraries. However, the second part of the loss is not differentiable and it needs to be computed differently. Concretely, the part not differentiable is the transformation from the 2D coordinates to the projected image via sampling, which is a discrete operation that would kill the gradients in the backward pass.

Figure 3.7 shows an illustration of the gradient propagation through the Projection Loss layer. As shown in the image, the only part that needs to be defined is the derivative of $\mathcal{L}_{IoU}$ with respect to the predicted 2D coordinates. Notice that, obviously, the gradients do not propagate to gates that do not depend on the weights of the network. This is why the gradients do not go to gates coming from ground truth values or to the 3D model points.

In order to obtain the derivative of $\mathcal{L}_{IoU}$ with respect to the 2D coordinates, we separate the gradient into two steps with the chain rule. In the first step, we will compute the gradient of the loss with respect to each pixel of the predicted projection. In the second step, we use an approximation to find the derivative of each pixel with respect to each 2D coordinate.

For the first step, we follow a similar approach as the outlined in [3] to obtain the derivative of $\mathcal{L}_{IoU}$ from Equation 3.11 with respect to the predicted image at each pixel $v \in V$. Being V the set of pixels of the images to compare, $X \in \{0,1\}^V$ the predicted mask and $Y \in \{0,1\}^V$ the ground truth mask:

$$\begin{aligned}\frac{\partial \mathcal{L}_{IoU}}{\partial X_v} &= \frac{\partial}{\partial X_v}\left[1 - \frac{I(X,Y)}{U(X,Y)}\right] \\ &= -\frac{\frac{\partial I(X,Y)}{\partial X_v}U(X,Y) - \frac{\partial U(X,Y)}{\partial X_v}I(X,Y)}{U(X,Y)^2}\end{aligned} \tag{3.12}$$

from Equations 3.9 and 3.10, we can easily derive:

$$\begin{aligned}\frac{\partial I(X,Y)}{\partial X_v} &= Y_v \\ \frac{\partial U(X,Y)}{\partial X_v} &= 1 - Y_v\end{aligned}$$

so, the derivative of $\mathcal{L}_{IoU}$ with respect to the predicted pixel $v \in V$ can be written as:

$$\frac{\partial \mathcal{L}_{IoU}}{\partial X_v} = \frac{I(X,Y)(1-Y_v) - U(X,Y)Y_v}{U(X,Y)^2} \tag{3.13}$$

since $Y_v \in \{0,1\}$, the equation can be simplified for the two possible cases as:

$$\frac{\partial \mathcal{L}_{IoU}}{\partial X_v} = \begin{cases} -\frac{1}{U(X,Y)} & \text{if } Y_v = 1 \\ \frac{I(X,Y)}{U(X,Y)^2} & \text{otherwise} \end{cases} \tag{3.14}$$

The second step to obtain the final local derivative is a bit more complex. Figure 3.8 shows an illustration of the problem. As we can see, due to the discrete nature of the operation, the derivative of it is zero for all points except the boundaries of the pixel. In this situation, the gradient from the $\mathcal{L}_{IoU}$ loss is multiplied by zero in all cases except the exact boundaries, therefore the effect of the loss is almost never propagated to the 2D coordinates and, consequently, to the weights of the network.

In order to obtain a better propagation of the error signal, we follow an approximation similar to the ones presented in [25] and [23], where the derivative of each pixel with respect to each coordinate is approximated such that the gradients can propagate backwards. In our case, we define the value of a pixels X_v as a linear intensity going from the middle of the adjacent pixels to the center of X_v. Therefore, at the very center, the intensity of the pixel is 1 and decreases linearly until the middle of the adjacent pixel. This is obviously just an approximation, but it will allow us to obtain a better derivative of each pixel with respect to each coordinate.

Then, considering that each pixel has the following relation with each (u_i, v_i) 2D coordinate:

$$X_v(u_i, v_i) = max(0, 1 - |m - u_i|)max(0, 1 - |n - v_i|) \tag{3.15}$$

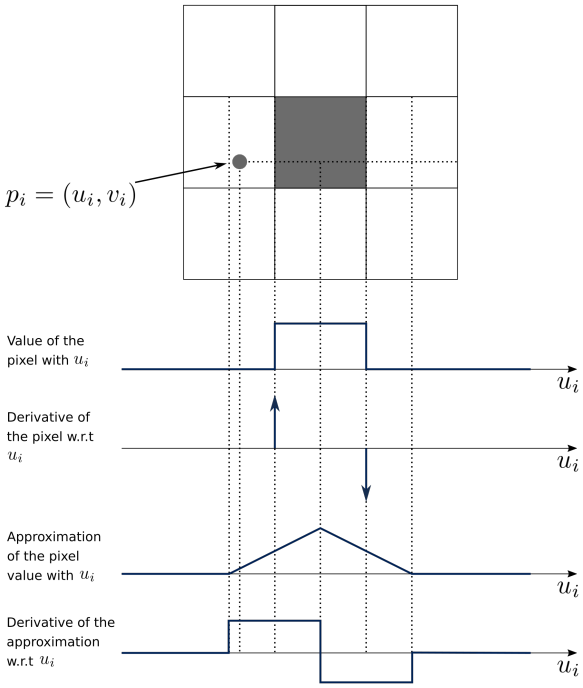


FIGURE 3.8: Illustration of the approximation of the derivative of each pixel with respect to the coordinates of the points. This is a similar approach to the one proposed in [25].

Where m and n are the centers of the pixel X_v. So we can define the derivative of each pixel X_v with respect to each point coordinate (u_i, v_i) as:

$$\frac{\partial X_v}{\partial u_i} = \begin{cases} max(0, 1 - |n - v_i|) & \text{if } u_i \leq m \\ -max(0, 1 - |n - v_i|) & \text{if } u_i > m \\ 0 & \text{if } |n - u_i| \geq 1 \end{cases} \tag{3.16}$$

where the same process can be used similarly for the derivation in v_i.

However, in practice, our projected images are binary masks and $X_v \in \{0, 1\}$. Therefore, in the backpropagation process, we will check if the evaluated pixel can change its value or not. Gradients for pixels containing already another coordinate point will be set to zero. Furthermore, pixels containing more than one point will only propagate to one of the points.

Therefore, the final local gradient of the gate for each 2D point of the predicted pose can be computed with the two partial derivatives described above. By applying chain rule, we can compute the derivative of the loss with respect to each 2D point for all pixels of the image. This can be implemented in CUDA as we will explain in the following section.

In the forward pass, we will store the projected images of the prediction and ground truth values, so that, in the backward pass, the above gradients can be computed. Notice that this is just the local gradient of the gate, in order to compute the gradient flowing out of the gate, we need to apply the chain rule and multiply the local gradient by the gradients coming into the gate as well.

Finally, notice that the inputs of the $\mathcal{L}_{IoU}$ Computation gate are the predicted 2D coordinates and the ground truth coordinates, which are expressed as tensors with shapes $[B, P, 3]$. The 3D vector for each point and batch corresponds to $[u, v, 1]$, where the first two dimensions contain the 2D coordinates and the third one is just used in previous operations to normalize the coordinates. Obviously, the gradient for the third dimension is set to zero, because it is constant at the input of the gate.

3.3.3 CUDA Implementation

As it was already stated, the second part of the Loss, corresponding to the $\mathcal{L}_{IoU}$ Computation in Figure 3.5, cannot be easily implement using the available operations in the TensorFlow library. However, since this operation is used many times during training, its implementation must be very efficient. This is the reason why this part is implemented using Parallel Programming in CUDA [34], whereby multiple threads can be executed directly in NVIDIA Graphics Process Units (GPUs).

CUDA C allows developers to perform parallel programming in NVIDIA CUDA architectures using a high-level programming language. Using this platform, we can implement new operations in C that leverages NVIDIA GPUs to solve complex computational problems in a highly parallelized way. The new operations can be then integrated into the already implemented networks. Most operations in the TensorFlow library are in fact implemented using CUDA and adding new operations to the library is fairly simple.

As it was explained in the previous section, the second part in the Figure 3.5 takes the 2D coordinates of two different poses, creates the projected image and computes the $\mathcal{L}_{IoU}$ loss. This can be seen as another graph operation or gate into the network, so we need to define its forward and backward pass as two different programs in CUDA C. The TensorFlow API[1] provides the framework to integrate the two programs as the forward and backward pass, and store all necessary tensors for the backpropagation step.

Forward Pass

Figure 3.9 shows an illustration of the forward pass implementation. In order to compute the loss, three different *kernels* are created to perform small tasks in a completely parallelized way. A kernel is a small code that runs in parallel in multiple GPU cores. As we can see in the illustration, we create three different kernels to draw the points of the predicted and ground truth poses, and count the intersection and union values.

The first thing to notice in the figure is the use of *blocks* and *threads*. This is nothing new in the Parallel Programming community, it is just the way it is organized in CUDA. In order to use parallel programming in a GPU, the kernels are launched in multiple threads to perform all the operations as parallel processes. Blocks are simply groups of multiple threads, which in more complex implementations of CUDA can be used to access certain shared memories quicker. In all the kernels implemented for this loss, all threads perform independently and the shared memory is

[1] TensorFlow documentation: Adding a New Op.
https://www.tensorflow.org/guide/extend/op

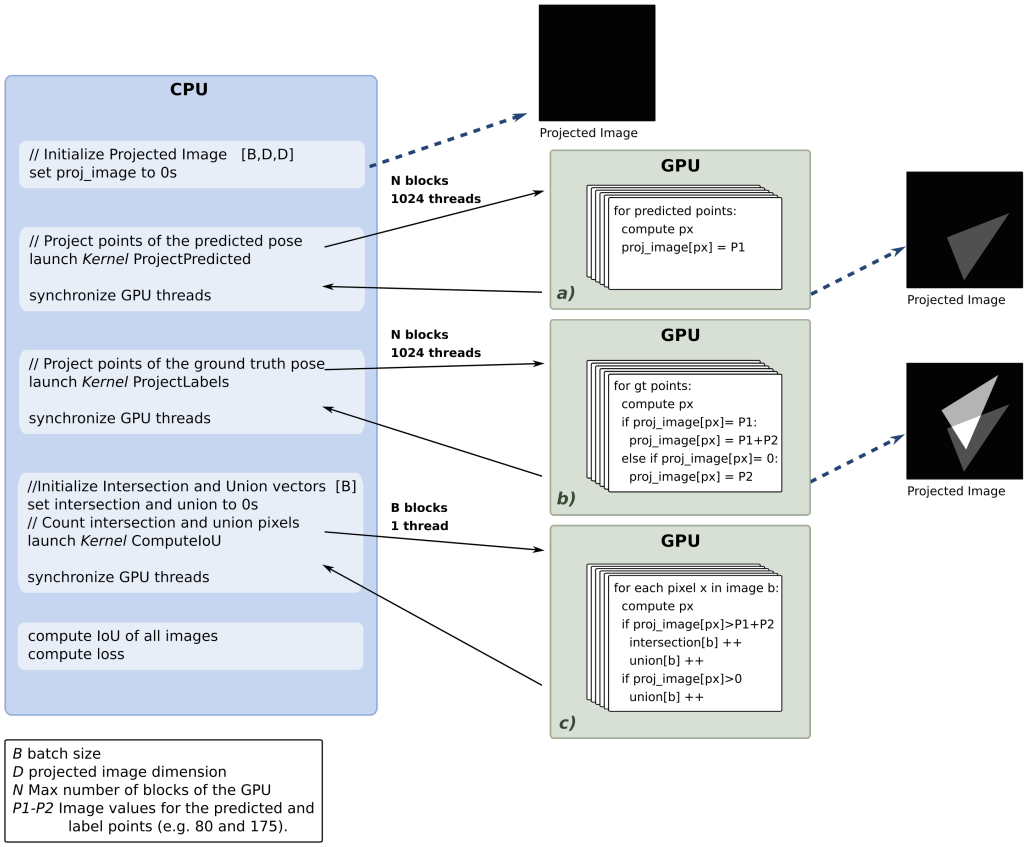


FIGURE 3.9: Forward pass implementation of the $\mathcal{L}_{IOU}$ computation in CUDA. The implementation uses three different kernels to parallelize three tasks in the GPU. a) Computes the point projections of the predicted pose. b) computes the point projections of the ground truth pose. c) counts the intersection and union pixels in each image. Notice that the constant $P1$ should be lower than $P2$.

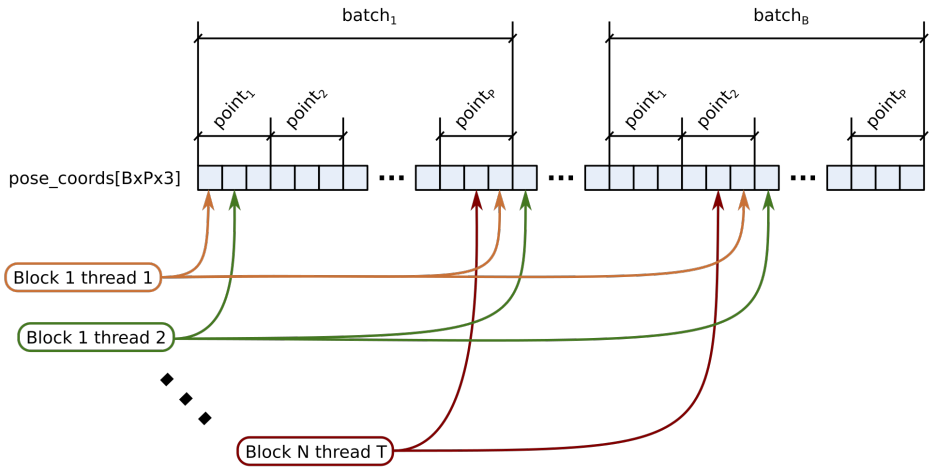


FIGURE 3.10: Illustration of the evaluation of the input pose vector using blocks and threads.

not used.
Using blocks and threads makes the iteration over an input array a bit different. Figure 3.10 shows an illustration example of how to use these calls to evaluate a long input array. In this example, the input vector is one of the inputs in the $\mathcal{L}_{IoU}$ computation gate (see Figure 3.5), and it is evaluated using N blocks of T threads each. Knowing the initial position and the total number of processes launched, each thread creates a loop to evaluate certain positions of the input vector, collectively exploring the whole data array.

In Figure 3.9, the first operation is the initialization of the projected images of the batch as a vector of zeros. The vector contains $B \cdot D \cdot D$ elements, where B is the number of batches and D is the size of the projected image. Once initialized, the result is a batch of black images like the one shown at the beginning of Figure 3.9.

The second operation is the call to the first kernel, which generates the projection of the predicted pose (named *ProjectPredicted* in the figure). This kernel is launched in the GPU using as many blocks as possible (with a maximum of $N = 65535$, which is the hardware limit) of $T = 1024$ threads each. Each thread iteratively draws points of the projected pose until all points have been evaluated. This kernel execution modifies the batch of projected images to contain the silhouette of the predicted pose as it is shown in the illustration.

The third operation is very similar to the previous one. In this case, the kernel generates the projection of the ground truth pose on the projected images. In order to do it, the kernel is called just like the previous one, using as many blocks as possible of 1024 threads each. In this case though, the kernel is a bit different and combines the previous projections with the new ones. Pixels containing both predicted and ground truth points are set to $P1 + P2$ and pixels containing just one of the poses are set to ether $P1$ or $P2$.

Using the images generated in the previous two operations, the third kernel basically computes the intersection and union values. This kernel is launched using as many blocks as elements in the batch. Each block contains a single thread, which counts the intersecting pixels (elements with a value of $P1 + P2$) and the union pixel

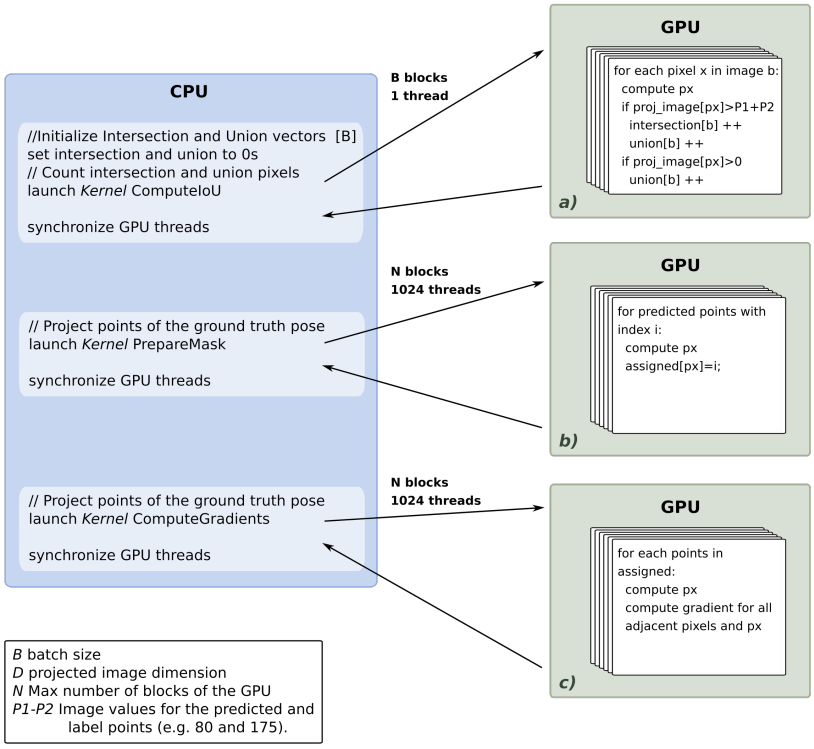


FIGURE 3.11: Backward pass implementation of the $\mathcal{L}_{IoU}$ computation in CUDA. The implementation uses two different kernels to parallelize two tasks in the GPU. a) Counts the intersection and union pixels of each projected image stored during the forward pass. b) Selects for each active point in the predicted projection, the coordinate that will propagate the error signal. c) Computes the gradient going backwards to each coordinate of the assigned array.

(any element bigger than 0) for each image. At the end of this execution, the intersection and union values for each image are completely defined. Using those values, the final loss can be easily computed. This final calculation is performed directly in the CPU, because the number of operations is not very large.
Notice that this third kernel could be implemented more efficiently, using more threads to truly leverage the power of the GPU. However, since a detailed study of computational time was not carried out, a more efficient implementation of this loss is left for future work.

Backward Pass

The backward pass implementation of the $\mathcal{L}_{IoU}$ computation operation is very similar to the forward pass. Figure 3.11 shows an illustration of this implementation in CUDA. As we can see, the backward pass also uses three kernels, one for the computation of the intersection and union pixels, another one to obtain the index of the pixels in the projected image and one to compute the gradient for each coordinate. Therefore, this implementation uses the projected images stored in memory during the forward pass.

The first operation performed during the backward pass is the computation of the intersection and union pixels. This is the same kernel used before in the forward pass and uses the same number of threads. Since the projected images during the forward pass are available, this kernel just counts for each image the number of pixels with value $P1 + P2$ (intersection) and the number of pixels with value grater than 0 (union).

The second operation prepares the mask for the gradient computation. The projected image is a binary image, so each pixel can be either be 1 or 0. Therefore, even though there can be multiple points in the area of one pixel, the value does not change. Therefore, in the backward pass, the gradients of that pixel will just propagate to one of the points (chosen randomly). This kernel selects from each point of the mask, the coordinate point that will be used to propagate the gradients.

Finally, the third operation computes the gradient defined earlier using the projected image and the intersection and union values. In this kernel, we check the conditions of each pixel individually to compute the gradient defined in Equation 3.16. The only coordinates that propagates the error signal are the ones computed in the kernel *PrepareMask*. This computation is pretty efficient, because, for each coordinate point, the computation of the gradients is just computed for the adjacent pixels.

3.4 Implementation Details

This section summarizes the main technical details of the network implementation. In the first part, the details of the network architecture, optimization process and weight initialization are given. In the second part, the main techniques used for data augmentation are explained.

3.4.1 Network Details

Most of the network was implemented in TensorFlow [1], except for part of the Projection Loss implementation that was directly programmed in CUDA C. The algorithm used to update weights was the Adam Optimizer [27] with a learning rate of $1 \cdot 10^{-4}$. The initialization for the weight of the fully connected layers was the Xavier initialization [18], using a normal distribution with 0 mean and standard deviation:

$$\sigma = \sqrt{\frac{2}{in + out}} \tag{3.17}$$

where in and out are the number of input and hidden neurons of the layer. The weights of the 13 convolutional layers in the feature extractor part are initialized using pre-trained weights. Since the backbone of the network has the same architecture as the well-known VGG network [42], trained weights on the ImageNet dataset [10] can be used as initial parameters.
Finally, all fully connected layers, except the linear ones, uses dropout of 15%

3.4.2 Data Augmentation

Even though the datasets used to train the NN are created synthetically, rendering scenes is still time consuming. In order to increase the size of dataset without any increment in the computational time, we use data augmentation as a first preprocessing step of the network.

The most simple way perform data augmentation is by randomly changing the brightness, contrast and sharpness of the image. The brightness and contrast will surely change in test time, so adding them in the training dataset will make our network more robust. Using random brightness and contrast, the network will be able to absorb more changes in the illumination of the scene. The sharpness of the image is an extra property that can be modified in the image to boost the performance of the network in test, helping to reduce the *reality gap* issue described in the previous chapter. We visually found that synthetic images looked more similar to the real ones increasing the sharpness. However, trying to find the perfect sharpness effect that would work for all images is a tedious task that must be done manually. Therefore, we add variations of it as a data augmentation change, so that the network can absorb variations of this kind in test.

Another strategy to increase the variability of a dataset in training is to add noise. During the preprocessing step, we can create a white noise image and add it to the input one with a certain weight. This creates a resulting image with what is commonly called *salt-and-pepper* noise. During training we add this noise to make the network more robust in test.

Finally, an obvious way to increase the dataset in training is rotating the image. However, in order to do it for the 6D pose regression task, the ground truth values have to be corrected accordingly. In our implementation, the image can be rotated by 0^o,90^o,180^o and 270^o, making the dataset four times bigger. In order to apply these changes to the labels, considering the camera in the origin of the coordinate system pointing straight down the z-axis, the new position and rotation of the objects can be computed as:

$$\mathbf{t}' = \mathbf{R_a} \cdot \mathbf{t} \tag{3.18}$$

$$\mathbf{q}' = \mathbf{q_a} \cdot \mathbf{q} \tag{3.19}$$

where $\mathbf{t}'$ and $\mathbf{q}'$ are the modified translation and rotation vectors. $\mathbf{R_a}$ and $\mathbf{q_a}$ depend on the rotation of the image:

$$\begin{cases} \mathbf{r_a} = (0, 0, -\frac{\pi}{2}) \rightarrow \mathbf{q_a} = (\frac{\sqrt{2}}{2}, 0, 0, -\frac{\sqrt{2}}{2}) & \text{if rotated } 90^o \text{ clockwise} \\ \mathbf{r_a} = (0, 0, -\pi) \rightarrow \mathbf{q_a} = (0, 0, 0, -1) & \text{if rotated } 180^o \\ \mathbf{r_a} = (0, 0, \frac{\pi}{2}) \rightarrow \mathbf{q_a} = (-\frac{\sqrt{2}}{2}, 0, 0, -\frac{\sqrt{2}}{2}) & \text{if rotated } 270^o \text{ clockwise} \end{cases} \tag{3.20}$$

and $\mathbf{R_a}$ can be obtained using $\mathbf{q_a}$ and equation 3.4.

Chapter 4

Results

This chapter evaluates the implementation described in Chapter 3, trained with the synthetic images created with the generation pipeline explained in Chapter 2. First, the main evaluation methods to compare the different variations are described. Second, an ablation study analyzing the main contributions of this work is performed. Finally, results on 6DSO are provided.

4.1 Evaluation Method

In order to evaluate the performance of the network with different hyperparameters, we perform two different evaluations: a quantitative evaluation and a qualitative evaluation. For the first one, we use the most common metrics in the literature, which are the ADD and the ADD-S score. For the qualitative evaluation instead, we visually compare the two poses, by rendering the predicted pose and comparing it to the ground truth values.

The ADD score, proposed in [21], computes the average 3D distances between the predicted pose and ground truth pose as:

$$m = \frac{1}{P} \sum_{\mathbf{x} \in \mathcal{M}} \|(\mathbf{R}\mathbf{x} + \mathbf{t}) - (\tilde{\mathbf{R}}\mathbf{x} + \tilde{\mathbf{t}})\| \tag{4.1}$$

where $\mathcal{M}$ is the set of points of the mesh and P is the total number of points. $\tilde{\mathbf{R}}$ and $\mathbf{R}$ are the predicted and ground truth rotation matrices respectively. $\tilde{\mathbf{t}}$ and $\mathbf{t}$ are the predicted and ground truth translation vectors.

The ADD score considers a predicted pose correct if the value m is smaller than a predefined threshold. In the literature, the threshold is set to a certain percentage of the 3D model diameter. For the quantitative evaluation of this work, the threshold set to 10% of the diameter, which is the most common value.

A variation of the ADD score is the ADD-S score, also defined in [21]. This matching score compares how two volumes match and it is more useful when comparing symmetric objects or objects with some ambiguous views. For the ADD-s score, the average is computed using the closest point between the two meshes:

$$m = \frac{1}{P} \sum_{\mathbf{x_1} \in \mathcal{M}} \min_{x_2 \in \mathcal{M}} \|(\mathbf{R}\mathbf{x_1} + \mathbf{t}) - (\tilde{\mathbf{R}}\mathbf{x_2} + \tilde{\mathbf{t}})\| \tag{4.2}$$

For the qualitative evaluation, for each trained model we compare some of the best predictions and some of the worst ones. With this comparison, we aim to analyze the reasons for the prediction errors and understand why some approach works better than another.

Finally, in the ablation study, we use three different datasets: training, validation and real. The training dataset contains only synthetic images and it has been enlarged with the data augmentation techniques explained in Chapter 3.4.2. The validation dataset also contains only synthetic images, but no data augmentation has been used. Finally, the real dataset contains only real images from the 6DSO dataset. In the ablation study we use half of the real dataset during training, so that we can visualize the effect of the domain randomization. Later, in Section 4.3, the other half of the dataset is used to give the final results in test.

4.2 Ablation Study

This part of the evaluation analyzes the two main contributions of the work separately. Concretely, the first section studies the effect of the domain randomization in the synthetic generation and the second section focuses on the effect of the Projection Loss on the final results.

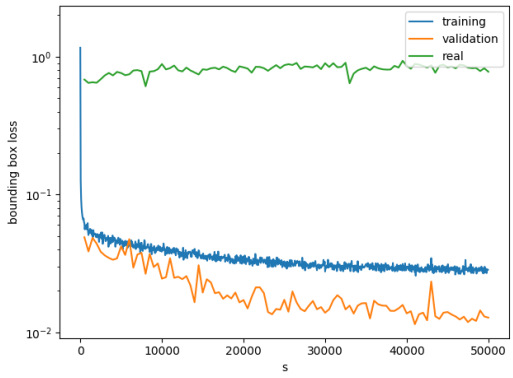
4.2.1 Effect of the domain randomization

As we already mentioned, the main problem of using synthetic data for training NNs is the *reality gap*. If the synthetic images are not exactly as the real ones, the network may not generalize well in test. The proposed solution in this work is to use domain randomization in the generation process. This section analyzes the effect of the different variations introduced in the generation process. For the analysis, we use the *Toy* object from the 6DSO dataset.

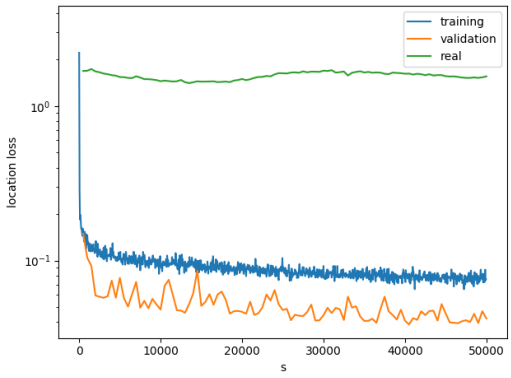
Evidence of the Reality Gap

Figure 4.1 shows results for a very naive synthetic generation of images. For this experiment, the generation system simply used solid backgrounds, fixed light conditions and no object variations. As we can observe, even though the results in validation are good, the results for real images are very poor. This figure perfectly illustrates the reality gap problem, where the network learns the pose regression task for synthetic images but not for real ones.

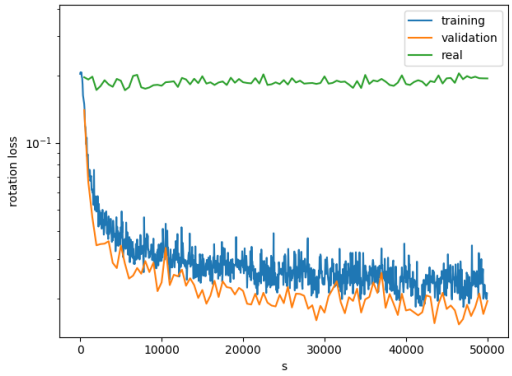
Analyzing the results closely, it can be seen that the network is not even able to locate the object within the image. The bounding box error in this experiment is over 0.6, which means that the box prediction used in the RoI pooling layer is not very good. Considering that the bounding box error is the Euclidean distance between the predicted and ground truth $\mathbf{b}$ vectors, it is clear that the crops of the RoI pooling layer are not even close to the object in most of the cases. Therefore, for real images, it is impossible to predict the rotation, because the features used in that branch are not the same as the ones used during training.



(A) Bounding Box Loss

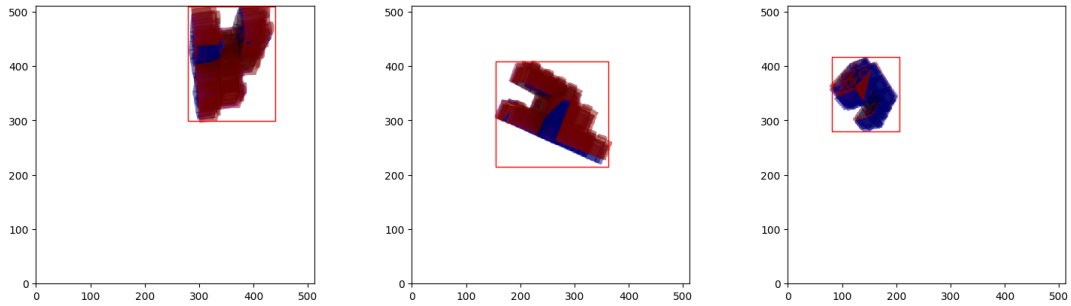


(B) Location Loss

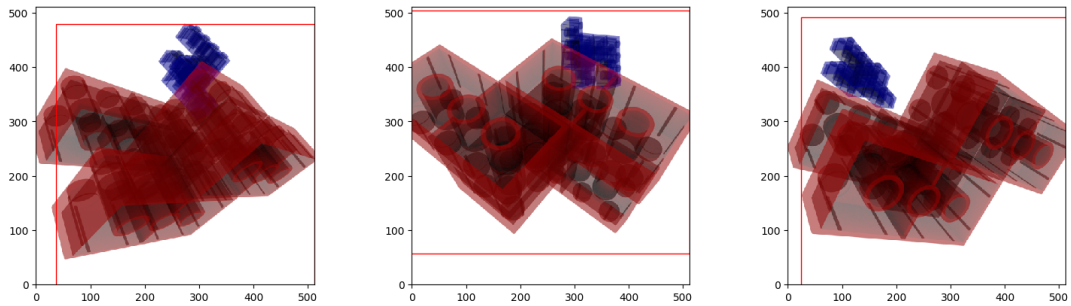


(C) Rotation Loss

FIGURE 4.1: Example of a naive synthetic generation. In this example, the synthetic images uses solid colors as background, fixed light conditions and fixed object properties.



(A) Test with synthetic images



(B) Test with real images

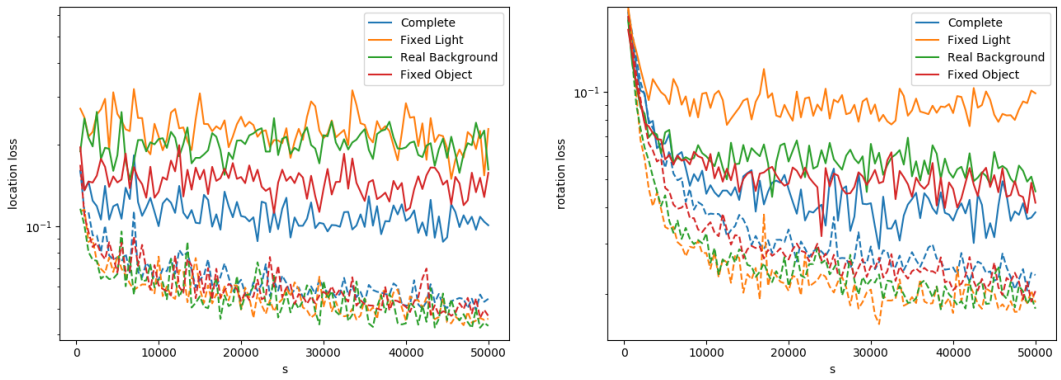
FIGURE 4.2: Results from the network in Figure 4.1. To visualize the results, two different objects at the predicted (red) and ground truth (blue) are created and rendered together, for synthetic test images (a) and real ones (b). The predicted bounding box is shown in red.

Note that the validation error is lower than the training error. This effect is due to the fact that the training dataset has been pre-processed with a lot of data augmentation. As it was explained in Chapter 3.4.2, images used for training are randomized using brightness and contrast variations, and salt-and-pepper noise.

Figure 4.2 shows examples of predictions with synthetic images and real ones. In order to visualize the two examples we synthetically generate two objects at the predicted (red) and ground truth (blue) poses. In the figure, it can be observed that, while the network accurately regresses the pose for synthetic images, it is not able to perform the task correctly for the real ones. In fact, the results for real images are always centered and in a random orientation, probably because the prediction of bounding box is completely incorrect and the features cropped with the RoI layer contain a lot of background. In the figure, we also show the bounding box prediction for each test instance, showing that it is almost perfect for synthetics images and it is completely erratic for the real ones.

Effect of the different domain randomization strategies

From the above experiment, it seems obvious that a naive generation of synthetic images is not working. In this work, we propose the use of domain randomization to close the reality gap. In order to analyze the effect of each randomization method described in Chapter 2.1.3, we perform several experiments partially removing some variations during the synthetic generation. Concretely, the different generation pipelines compared are:



(A) Location Loss (B) Rotation Loss

FIGURE 4.3: Comparison of different variations of the domain randomization process. The solid and dashed lines show the results for the real and validation datasets respectively.

- **Complete**: This generation pipeline creates completely randomized datasets. For the background, it uses random images from the PASCAL VOC dataset[12], from the DTD dataset[9] and from a small collection of real images of the background. For the illumination, it uses a combination of spotlights and a global environmental light, randomly changing the position and intensity of the different sources of light. Finally, this generation system also introduces variations on the object itself, slightly changing its shape and color.
- **Fixed Light**: This second synthetic generation system uses the same variations as *Complete* except for the lighting conditions. Instead of using a combination of spotlights and an environmental light, the synthetic images are created just with a constant environmental light.
- **Real Background**: This generation pipeline uses as background only real images, randomly selecting crops of the same background it will be seen in test. The pipeline introduces the same variations in the lighting and the object as the *Complete* one.
- **Fixed Object**: Finally, this is the same generation pipeline as the *Complete* one, but fixing the object conditions. In the synthetic images generated with this setup, the object does not vary its shape and color, using just the nominal values.

Figure 4.3 shows the results for the validation and real datasets of the different generation pipelines described above. As it can be observed, the domain randomization clearly makes the reality gap smaller. The more randomization is added to the training dataset, the smaller the difference between the synthetic and real images is in test. While adding variations slightly increases the error for the validation dataset, the error for the real images clearly drops as more randomization is added.

Table 4.1 summarizes the ADD and ADD-S for all the generation pipelines. As it can be observed, the effect of the domain randomization is crucial to obtain good results with real images.

	SYNTHETIC IMAGES		REAL IMAGES	
	ADD	ADD-S	ADD	ADD-S
Complete	67.8	71.9	43.9	48.7
Fixed Light	78.4	81.3	21.7	29.6
Real Background	81.3	84.7	24.3	27.0
Fixed Object	72.5	76.9	41.2	44.7
Without Randomization	81.3	84.7	0.0	0.0

TABLE 4.1: Performance with the different synthetic generation pipelines.

4.2.2 Effect of the Projection Loss

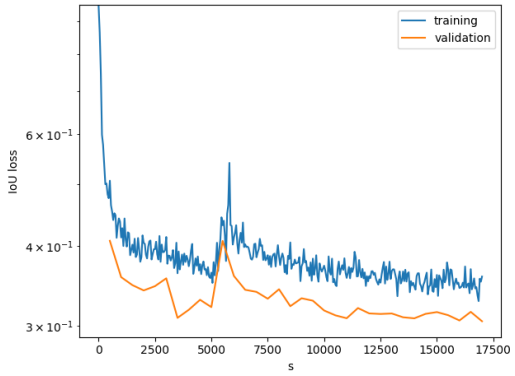
In this section, the effect of the projection loss on different settings is studied. In order to first ensure that the backpropagation defined in Chapter 3.3.3 is updating the weights correctly, the training shown in Figure 4.4 is performed. In this experiment, we train the network exclusively using the Projection Loss and the bounding box loss, finding that it actually improves the IoU of the projected and ground truth poses.

However, in Figure 4.4 we can see that the loss only achieves some improvement on the location prediction, but not on the rotation one. This makes a lot of sense, because, while the location of the object can be more or less estimated only comparing its silhouette, predicting the rotation with that information alone is much more complex. Intuitively, without prior knowledge, learning the different view of an object only using the silhouettes seems extremely difficult. We believe that this loss will help in the rotation prediction after the main features of the object have been learned. Therefore, the Projection Loss needs the rotation loss to be useful and it cannot work by itself.
Note that the bounding box loss has no effect on the prediction of the location and rotation. It was added in the training because it is necessary to perform the correct crops in the RoI pooling layer.

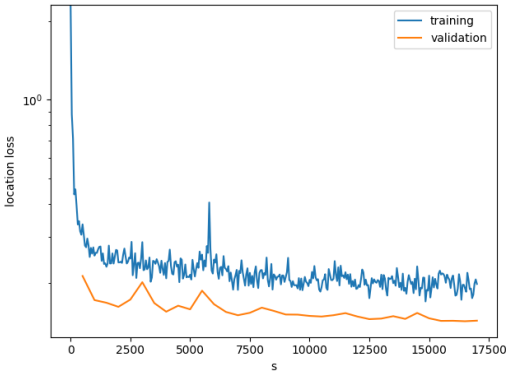
In order to compare the results with the Projection Loss and without it, we train the network for the *Symmetric* object of the 6DSO dataset. This object has multiple ambiguous views, so the effect of the loss should be clearly visible in this case. The Projection Loss should help the network to decide one of the two possible symmetric orientations and prevent the update to erroneous poses. Figure 4.5 shows the experiment for the validation dataset over 100 epochs.

As it can be seen in Figure 4.5, there is a small improvement in the rotation error, which is smaller for the network using the projection loss. However, the difference between the two rotation errors is pretty small and only affecting the rotation in the last steps of the training. This effect makes a lot of sense, because the Projection Loss is not actually able to find the correct ground truth value for ambiguous cases, but it just helps the network to decide one of the possible orientations. The ambiguous views are obviously impossible to determine, so the network using the projection loss just chooses one possible solution, but not necessarily the correct one.

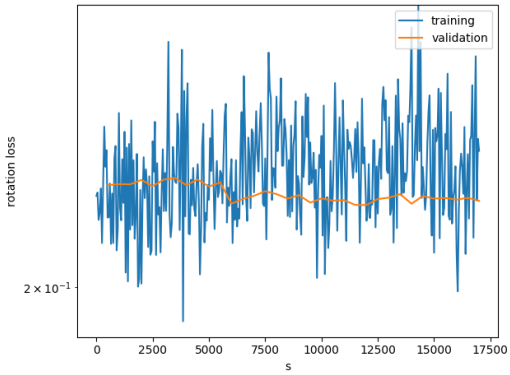
From the trained network in Figure 4.5, we can perform a qualitative study with some real images. Figure 4.6 shows some predictions of the two networks for ambiguous views. As we can see in the first two examples, while the network without the Prediction Loss failed to predict one of the correct view, the one with the new



(A) IoU Loss

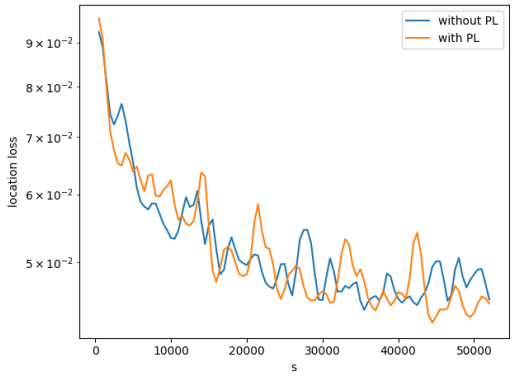


(B) Location Loss

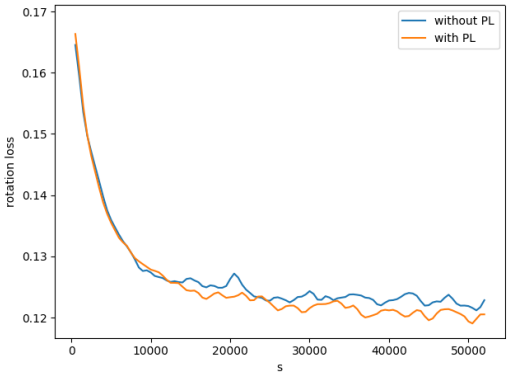


(C) Rotation Loss

FIGURE 4.4: Results for the network trained exclusively using the Projection Loss and the bounding box loss. The object used here is the *Toy* object defined in Section 2.2.2



(A) Location Loss



(B) Rotation Loss

FIGURE 4.5: Training of the network for the *Symmetric* object defined in Section 2.2.2. The curves were obtained with the validation dataset.

(A) Original Image (B) Without Projection Loss (C) With Projection Loss

FIGURE 4.6: Examples of predictions for the two networks shown in Figure 4.5. The first two examples are cases where the projection loss helped to correctly decide one of the correct view. In the third example, the network without the projection loss actually predicted one of the correct views, but the network with the projection loss did not help and performed slightly worse.

loss estimated correctly one of the correct poses. However, as we can see in the third example, in some cases the network without the loss is able to regress one valid orientation and the new loss does not help.

In the qualitative evaluation we notice an issue with the projection loss. Some ambiguous views cannot be corrected with it, because the projected silhouettes do not contain enough information of the orientation. The *Symmetric* object is a clear example of this problem, because the projected image for this object is just a square in many cases and does not provide useful information. In those cases, the IoU metric can be very similar even though the predicted and ground truth orientations are completely different, so the effect of the new loss loses importance.

Performing a quantitative study of the two networks, we obtain the results shown in Table 4.2. As it can be observed, the Projection Loss improves the results for both metrics, but more clearly for the ADD-S measure. On the one hand, the first metric does not take into account the symmetries, so the score for both networks is pretty

	ADD	ADD-S
Without Projection Loss	34.5	51
With Projection Loss	36.2	60.7

TABLE 4.2: Comparison between the two networks shown in 4.5 for real images.

low. Even though the new loss improves a bit the results even for this metric, the difference is very small. The reason for this is that even perfectly picking one of the valid options, the ground truth orientation could be completely the opposite, making ADD score lower.
On the other hand, the ADD-S metric is more or less like comparing volumes at the predicted and ground truth poses. With this measure, the effect of the Projection Loss is much more noticeable, because just predicting one of the valid orientation results in a low distance between the closest points. Therefore, if the two volumes at the predicted and ground truth poses are exactly the same, the ADD-S will be high. The results shown in that table evidences the benefits of the new loss for ambiguous cases.

4.3 Results on the created dataset

Finally, we evaluate our system on the three objects from the 6DSO dataset. The results for the two metrics on the three objects are shown in Table 4.3. For the two objects that are not symmetric, the network can predict the pose with an error smaller than 10% of the diameter in 44% and 69% of the cases for the *Toy* object and the *Box* respectively. The symmetric object was already studied in the above section and the network would predict a valid pose 60% of the time. However, looking at the predicted images, it seems clear that using some refinement method as many do in the literature [36, 26, 31], we could improve quite a lot the results.

	ADD	ADD-S
Toy	43.9	48.7
Box	69.0	74.3
Symmetric	36.2	60.7

TABLE 4.3: Final results for the 6DSO dataset.

Figure 4.7 and 4.8 show the three best predictions and the three worst predictions of the network for each object. From the obtained results, we can clearly state that the Box object is the one producing the best results. Even the worst results are fairly accurate and could be easily improved with some refinement method. This is probably due to the fact that the box is a simple object and that it has a serigraph. Even though the synthetic images of the box are visibly different than the real ones, since the serigraph is added in the scene as an image texture, the final render contains many real features that can be used by the network. In order to achieve better generalization, during the synthetic generation we also introduced small variations on the position, reflectivity and color intensity of the serigraph.

Contrarily, the worst results are clearly obtained for the *Toy* object. Even though some views are always predicted correctly, this object contains some other views that the network cannot regress properly. The main hypothesis is that this object has

a very complex shape, which creates many strange shades on the object itself and on the ground. Probably, these shades projected on the same object cause variations on the appearance that are complex to reproduce with the render. Similary, the shades on the ground create fake silhouettes very similar to the object, which probably introduces some error in the bounding box prediction.
In future works, a possible solution could be the one presented in [47], which improves the domain randomization using random textures on the objects and flying distractors on the background. With the random textures they introduce many unrealistic variations on the object itself, making the network generalize better. With the flying distractors, they add on the background shapes that are similar to the object, which forces the bounding box regression to be more robust.

Finally, the *Symmetric* object is predicted fairly accurately in most of the tests, but the network cannot be as precise as with the *Box*. The hypothesis here is that the symmetries still make the optimization process inefficient. Most of the predictions are slightly rotated on the symmetric axis, which means that the projection loss is not enough to correct the issue. As it was stated in Chapter 3, for completely symmetric objects, this problem can be easily corrected by modifying the quaternion labels to limit the pose to just some part of the rotation space. However, this is not applicable when the objects are not entirely symmetric, containing some small details breaking the symmetries.
For future works, we propose to extend the projection loss to compare complete rendered projections and not simply the silhouettes. Using a differentiable rendering layer as the one proposed in [25], the Projection Loss could be extended to compare the two poses with full 2D projection images.

(A) Toy

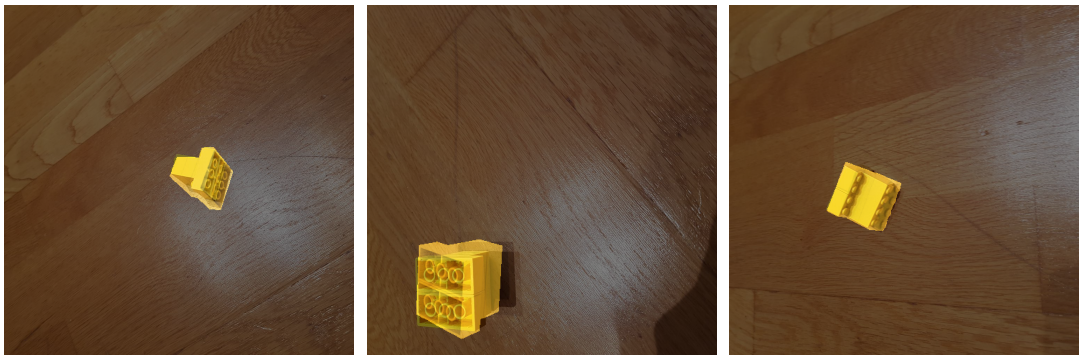
(B) Box

(C) Symmetric

FIGURE 4.7: Best three predictions for each object of the 6DSO

(A) Toy

(B) Box



(C) Symmetric

FIGURE 4.8: Worst three predictions for each object of the 6DSO

Chapter 5

Conclusion

5.1 Summary of Contributions

The preceding chapters have presented novel approaches for the 6D pose regression task using Deep Learning. The main contributions of the work were a generation pipeline to create synthetic datasets for training, a visual interface to manually label images for the pose regression task, a new test dataset with manually annotated real images and a new architecture with a novel loss function to predict the rotation and location of objects from RGB images.

First, the problem of obtaining sufficiently large datasets for optimizing deep models was addressed using synthetic data for training. The work presented a generation system to create fairly realistic datasets in computationally feasible times. In order to achieve good results with real images and overcome the so called *reality gap* problem, domain randomization techniques were used during the synthetic generation process.

Second, the work contributed with a visual interface to quickly find the pose of objects in real images. Even though the task of manually annotating real data is still very slow, the visual interface created in this work can simplify it and speed up the process. Furthermore, the work also contributed with a new dataset with three different objects, which could be extended in future works.

Finally, a new multi-task Neural Network architecture was proposed for the 6D pose regression task. The proposed network separately predicted the localization of the object within the image, the regression of the 3D location with respect to the camera and the regression of the 3D rotation with respect to a reference orientation. In order to improve the results for ambiguous and symmetric views, a new loss function was presented, which helped the network to better match the predicted and ground truth projected silhouettes.

All these different approaches have proven to be effective at obtaining the pose of objects from simple RGB images. In the evaluation section, it was proven that it is possible to infer the pose of real objects with networks that were trained exclusively with synthetic images. The value of the domain randomization methods were demonstrated, obtaining fairly close results in validation with synthetic images and in test with real ones. Furthermore, the benefits of the new loss function proposed to deal with ambiguous views were proven. Even though the results were far from perfect, with the proper refinement methods, the 6D pose regression could be implemented in real automation applications in a near future.

5.2 Directions for Future Work

This section outlines several promising directions for future work and potential extensions of the work presented in the preceding chapters.

Synthetic generation with random textures and flying distractors

As it was shown in the evaluation chapter, the results with real images are still not as good as the ones obtained with synthetic ones. Following the work proposed in [47], the synthetic generation pipeline could be extended to incorporate random textures for the objects and flying distractors on the background. On the one hand, the use of random textures for the objects would probably make the network more robust to variation in color, shape and illumination changes. We propose to use the DTD dataset[9] as new textures for the objects in the generation pipeline.
On the other hand, the presence of flying distractors would help the network to ignore the shades of the object projected on the ground. For future works we propose to improve the randomization methods by adding the silhouettes of the object itself on the background image.

Generative Adversarial Networks to add realism

A promising approach to tackle the reality gap problem is the use of GANs to add realism to the synthetic images. As it is shown in [41], more realism to synthetic images can be added using adversarial networks trained with unlabeled real data. We propose to use as the training dataset, a combination of synthetic images with domain randomization and realistic synthetic images improved with a GANs.

Render Loss

Even though the Projection Loss improved the results of the initial architecture, the ambiguous views still produce erratic predictions. The new loss can help to improve the prediction for some ambiguous views, but comparing only the silhouettes is not useful for all cases. In some situations, the foreground mask does not contain the important information to lead the optimization to the correct direction. We propose to extend the Projection Loss to compare not only the silhouettes, but the complete projected image. We propose to adapt the differentiable render presented in [25] for the pose regression task.

In addition, our implementation of the Projection Loss for the backward pass simplified the gradients of the projected images to just nearby projected coordinates. As it is shown in [25], the differentiable render can compute the influence of each 2D coordinate for all pixels of the projected image. We propose to update the backpropagation of the Projection Loss with this new approximation.

Multi view 6D pose regression

Training deep models using synthetic images expands the possibilities for the 6D pose regression task. With the generation pipeline, the images for training can be

created effortlessly for one or multiple cameras at the same time. We propose to extend the proposed architecture to work with multiple views at the same time. Using multiple cameras would probably eliminate most of the ambiguous views and it would surely achieve much better results. Future work lines could combine multiple architectures like the one proposed in this work to create a multi-view 6D pose regression system.

Bibliography

[1] Martín Abadi et al. "TensorFlow: A system for large-scale machine learning". In: (May 27, 2016). arXiv: 1605.08695v2 [cs.DC].

[2] Simon L. Altmann. *Rotations, Quaternions, and Double Groups (Dover Books on Mathematics)*. Dover Publications, 2013. ISBN: 9780486317731.

[3] Md Atiqur Rahman and Yang Wang. "Optimizing Intersection-Over-Union in Deep Neural Networks for Image Segmentation". In: vol. 10072. Dec. 2016, pp. 234–244. ISBN: 978-3-319-50834-4. DOI: 10.1007/978-3-319-50835-1_22.

[4] *Blenchmark Hardware Performance in Blender 3D*. http://blenchmark.com. Accessed: 2018-03-14.

[5] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation. Blender Institute, Amsterdam. URL: http://www.blender.org.

[6] Eric Brachmann et al. "Learning 6D Object Pose Estimation Using 3D Object Coordinates". In: vol. 8690. Sept. 2014, pp. 536–551. DOI: 10.1007/978-3-319-10605-2_35.

[7] G. Bradski. "The OpenCV Library". In: *Dr. Dobb's Journal of Software Tools* (2000).

[8] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision in C++ with the OpenCV Library*. 2nd. O'Reilly Media, Inc., 2013. ISBN: 1449314651, 9781449314651.

[9] M. Cimpoi et al. "Describing Textures in the Wild". In: *Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*. 2014.

[10] J. Deng et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[11] Thanh-Toan Do et al. "Deep-6DPose: Recovering 6D Object Pose from a Single RGB Image". In: (Feb. 28, 2018). arXiv: 1802.10367v1 [cs.CV].

[12] M. Everingham et al. *The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results*. URL: http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[13] Philipp Fischer et al. "FlowNet: Learning Optical Flow with Convolutional Networks". In: (Apr. 26, 2015). arXiv: 1504.06852v2 [cs.CV].

[14] Adrien Gaidon et al. "Virtual Worlds as Proxy for Multi-Object Tracking Analysis". In: (May 20, 2016). arXiv: 1605.06457v1 [cs.CV].

[15] Adrien Gaidon et al. "Virtual Worlds as Proxy for Multi-Object Tracking Analysis". In: (May 20, 2016). arXiv: 1605.06457v1 [cs.CV].

[16] Andreas Geiger. "Are We Ready for Autonomous Driving? The KITTI Vision Benchmark Suite". In: *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. CVPR '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 3354–3361. ISBN: 978-1-4673-1226-4. URL: http://dl.acm.org/citation.cfm?id=2354409.2354978.

[17] Ross Girshick. "Fast R-CNN". In: (Apr. 30, 2015). arXiv: 1504.08083v2 [cs.CV].

[18] Xavier Glorot and Y Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Journal of Machine Learning Research - Proceedings Track* 9 (Jan. 2010), pp. 249–256.

[19] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. 2nd ed. Cambridge University Press, 2004. DOI: 10.1017/CBO9780511811685.

[20] Kaiming He et al. "Mask R-CNN". In: (Mar. 20, 2017). arXiv: 1703.06870v3 [cs.CV].

[21] Stefan Hinterstoisser et al. "Model Based Training, Detection and Pose Estimation of Texture-less 3D Objects in Heavily Cluttered Scenes". In: *Proceedings of the 11th Asian Conference on Computer Vision - Volume Part I*. ACCV'12. Daejeon, Korea: Springer-Verlag, 2013, pp. 548–562. ISBN: 978-3-642-37330-5. DOI: 10.1007/978-3-642-37331-2_42. URL: http://dx.doi.org/10.1007/978-3-642-37331-2_42.

[22] Tomas Hodan et al. "T-LESS: An RGB-D Dataset for 6D Pose Estimation of Texture-less Objects". In: (Jan. 19, 2017). arXiv: 1701.05498v1 [cs.CV].

[23] Max Jaderberg et al. "Spatial Transformer Networks". In: (June 5, 2015). arXiv: 1506.02025v3 [cs.CV].

[24] Stephen James and Edward Johns. "3D Simulation for Robot Arm Control with Deep Q-Learning". In: (Sept. 13, 2016). arXiv: 1609.03759v2 [cs.RO].

[25] Hiroharu Kato, Yoshitaka Ushiku, and Tatsuya Harada. "Neural 3D Mesh Renderer". In: (Nov. 20, 2017). arXiv: 1711.07566v1 [cs.CV].

[26] Wadim Kehl et al. "SSD-6D: Making RGB-based 3D detection and 6D pose estimation great again". In: (Nov. 27, 2017). arXiv: 1711.10006v1 [cs.CV].

[27] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: (Dec. 22, 2014). arXiv: 1412.6980v9 [cs.LG].

[28] N. Koenig and A. Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 3. 2004, 2149–2154 vol.3. DOI: 10.1109/IROS.2004.1389727.

[29] Rigas Kouskouridas et al. "Latent-Class Hough Forests for 6 DoF Object Pose Estimation". In: (Feb. 3, 2016). arXiv: 1602.01464v1 [cs.CV].

[30] S. M. LaValle. *Planning Algorithms*. Available at http://planning.cs.uiuc.edu/. Cambridge, U.K.: Cambridge University Press, 2006.

[31] Yi Li et al. "DeepIM: Deep Iterative Matching for 6D Pose Estimation". In: (Mar. 31, 2018). arXiv: 1804.00175v2 [cs.CV].

[32] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: (May 1, 2014). arXiv: 1405.0312v3 [cs.CV].

[33] Wei Liu et al. "SSD: Single Shot MultiBox Detector". In: (Dec. 8, 2015). DOI: 10.1007/978-3-319-46448-0_2. arXiv: 1512.02325v5 [cs.CV].

[34] John Nickolls et al. "Scalable Parallel Programming with CUDA". In: *Queue* 6.2 (Mar. 2008), pp. 40–53. ISSN: 1542-7730. DOI: 10.1145/1365490.1365500. URL: http://doi.acm.org/10.1145/1365490.1365500.

[35] *OpenGL*. Khronos Group. URL: https://www.opengl.org.

[36] Mahdi Rad and Vincent Lepetit. "BB8: A Scalable, Accurate, Robust to Partial Occlusion Method for Predicting the 3D Poses of Challenging Objects without Using Depth". In: (Mar. 31, 2017). arXiv: 1703.10896v2 [cs.CV].

[37] Mahdi Rad, Markus Oberweger, and Vincent Lepetit. "Feature Mapping for Learning Fast and Accurate 3D Pose Inference from Synthetic Images". In: (Dec. 11, 2017). arXiv: 1712.03904v2 [cs.CV].

[38] Joseph Redmon and Ali Farhadi. "YOLO9000: Better, Faster, Stronger". In: (Dec. 25, 2016). arXiv: 1612.08242v1 [cs.CV].

[39] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". In: (June 8, 2015). arXiv: 1506.02640v5 [cs.CV].

[40] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: (June 4, 2015). arXiv: 1506.01497v3 [cs.CV].

[41] Ashish Shrivastava et al. "Learning from Simulated and Unsupervised Images through Adversarial Training". In: (Dec. 22, 2016). arXiv: 1612.07828v2 [cs.CV].

[42] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: (Sept. 4, 2014). arXiv: 1409.1556v6 [cs.CV].

[43] Christian Szegedy et al. "Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning". In: (Feb. 23, 2016). arXiv: 1602.07261v2 [cs.CV].

[44] Bugra Tekin, Sudipta N. Sinha, and Pascal Fua. "Real-Time Seamless Single Shot 6D Object Pose Prediction". In: (Nov. 24, 2017). arXiv: 1711.08848v4 [cs.CV].

[45] Josh Tobin et al. "Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World". In: (Mar. 20, 2017). arXiv: 1703.06907v1 [cs.RO].

[46] E. Todorov, T. Erez, and Y. Tassa. "MuJoCo: A physics engine for model-based control". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012, pp. 5026–5033. DOI: 10.1109/IROS.2012.6386109.

[47] Jonathan Tremblay et al. "Training Deep Networks with Synthetic Data: Bridging the Reality Gap by Domain Randomization". In: (Apr. 18, 2018). arXiv: 1804.06516v3 [cs.CV].

[48] Apostolia Tsirikoglou et al. "Procedural Modeling and Physically Based Rendering for Synthetic Data Generation in Automotive Applications". In: (Oct. 17, 2017). arXiv: 1710.06270v2 [cs.CV].

[49] *Unity 3D*. Unity Technologies. URL: https://unity3d.com/.

[50] Yu Xiang et al. "PoseCNN: A Convolutional Neural Network for 6D Object Pose Estimation in Cluttered Scenes". In: (Nov. 1, 2017). arXiv: 1711.00199v3 [cs.CV].

[51] Xucong Zhang et al. "Appearance-Based Gaze Estimation in the Wild". In: (Apr. 11, 2015). DOI: 10.1109/CVPR.2015.7299081. arXiv: 1504.02863v1 [cs.CV].